An abridged version of this paper appears in the proceedings of the 37th International Cryptology Conference—CRYPTO 2017. This is the full version.

Non-Malleable Codes for Space-Bounded Tampering

Sebastian Faust*,[1], Kristina Hostáková*,[1], Pratyay Mukherjee†,[2], and Daniele Venturi‡,[3]

[1] *Ruhr-Universität Bochum, Bochum, Germany*
[2] *Visa Research, Palo Alto, USA*
[3] *Sapienza University of Rome, Rome, Italy*

June 6, 2017

Abstract

Non-malleable codes—introduced by Dziembowski, Pietrzak and Wichs at ICS 2010—are key-less coding schemes in which mauling attempts to an encoding of a given message, w.r.t. some class of tampering adversaries, result in a decoded value that is either identical or unrelated to the original message. Such codes are very useful for protecting arbitrary cryptographic primitives against tampering attacks against the memory. Clearly, non-malleability is hopeless if the class of tampering adversaries includes the decoding and encoding algorithm. To circumvent this obstacle, the majority of past research focused on designing non-malleable codes for various tampering classes, albeit assuming that the adversary is unable to decode. Nonetheless, in many concrete settings, this assumption is not realistic.

In this paper, we explore one particular such scenario where the class of tampering adversaries naturally includes the decoding (but not the encoding) algorithm. In particular, we consider the class of adversaries that are restricted in terms of memory/space. Our main contributions can be summarized as follows:

- We initiate a general study of non-malleable codes resisting space-bounded tampering. In our model, the encoding procedure requires large space, but decoding can be done in small space, and thus can be also performed by the adversary. Unfortunately, in such a setting it is impossible to achieve non-malleability in the standard sense, and we need to aim for slightly weaker security guarantees. In a nutshell, our main notion (dubbed *leaky space-bounded non-malleability*) ensures that this is the best the adversary can do, in that space-bounded tampering attacks can be simulated given a small amount of leakage on the encoded value.
- We provide a simple construction of a leaky space-bounded non-malleable code. Our scheme is based on any Proof of Space (PoS)—a concept recently put forward by Ateniese *et al.* (SCN 2014) and Dziembowski *et al.* (CRYPTO 2015)—satisfying a

*Funded by the Emmy Noether Program FA 1320/1-1 of the German Research Foundation (DFG).

†Part of this work was done when the author was a Post-doctoral Employee at University of California, Berkeley, supported in part from DARPA/ARL SAFEWARE Award W911NF15C0210, AFOSR Award FA9550-15-1-0274, NSF CRII Award 1464397, AFOSR YIP Award and research grants by the Okawa Foundation and Visa Inc. The views expressed are those of the author and do not reflect the official policy or position of the funding agencies.

‡Partially supported by the European Unions Horizon 2020 research and innovation programme, under grant agreement No. 644666, and by CINI Cybersecurity National Laboratory within the project FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks (`www.filierasicura.it`), funded by CISCO Systems Inc. and Leonardo SpA.

variant of soundness. As we show, our paradigm can be instantiated by extending the analysis of the PoS construction by Ren and Devadas (TCC 2016-A), based on so-called stacks of localized expander graphs.

- Finally, we show that our flavor of non-malleability yields a natural security guarantee against memory tampering attacks, where one can trade a small amount of leakage on the secret key for protection against space-bounded tampering attacks.

Contents

1 Introduction

Non-malleable codes (NMC) [21] were originally proposed by Dziembowski, Pietrzak and Wichs [21] in 2010 and have since been studied intensively by the research community (see, e.g., [35, 25, 13, 26, 33, 1, 10] for some examples). Such codes are an extension of the concept of error correction and detection and can guarantee the integrity of a message in the presence of tampering attacks when error correction/detection may not be possible. Informally, a non-malleable code $(\mathsf{Encode}, \mathsf{Decode})$ guarantees that a codeword modified via an algorithm A, from some class $\mathcal{A}$ of allowed tampering attacks,[1] either encodes the original message, or a completely unrelated value. Notice that non-malleable codes do not need to correct or detect errors. This relaxation enables us to design codes that resist much broader tampering classes $\mathcal{A}$ than what is possible to achieve for error correcting/detecting codes. As an illustrative example, it is trivial to construct non-malleable codes for the class of constant tampering functions; that is, e.g., functions that replace the codeword by a different but valid codeword. Clearly, the output of a constant tampering function is independent of the original encoded message, and hence satisfies the non-malleability property. On the other hand, it is impossible to achieve error correction/detection against such tampering classes, as by definition valid codewords do not contain errors.

Applications of non-malleable codes. The fact that non-malleable codes can be built for broader tampering classes makes them particularly attractive as a mechanism for protecting the memory of physical devices from tampering attacks [8, 3]. To protect a cryptographic functionality $\mathcal{F}$ against tampering with respect to a class of attacks $\mathcal{A}$ applied to a secret key κ that is stored in memory, we can proceed as follows. Instead of storing κ directly in memory, we use a non-malleable code for $\mathcal{A}$, and store the codeword $c \leftarrow \mathsf{Encode}(\kappa)$. Thus, each time when $\mathcal{F}$ wants to access κ, we first decode $\tilde{\kappa} = \mathsf{Decode}(c)$, and, only if $\mathsf{Decode}(c) \neq \bot$, we run $\mathcal{F}(\tilde{\kappa}, \cdot)$ on any input of our choice. Intuitively, as long as the adversary can only apply tampering attacks from the class $\mathcal{A}$, non-malleability of $(\mathsf{Encode}, \mathsf{Decode})$ guarantees that any tampering results into a key that is unrelated to the original key, and hence the output of $\mathcal{F}$ does not reveal information about the original secret key. For further discussion on the application of non-malleable codes to tamper resilience we refer the reader to [21].

The tampering class $\mathcal{A}$. It is impossible to have codes that are non-malleable for *all* possible (efficient) tampering algorithms A. For instance, if $\mathcal{A}$ contains the composition of Encode and Decode, then given a codeword c the adversary can apply a tampering algorithm A that first decodes c to get the encoded value x; then, e.g., it flips the first bit of x to obtain $\tilde{x}$, and re-encodes $\tilde{x}$. Clearly, such an attack results into $\tilde{x}$ that is related to the original value x, and non-malleability is violated. A major research direction is hence to design non-malleable codes for broad classes of tampering attacks that exclude the above obvious attacks. Prominent examples are bit-wise tampering [21], where the adversary can modify each bit of the codeword individually, split-state tampering [2], where the codeword consists of two (possibly large) parts that can be tampered with individually, and tampering functions with bounded complexity [27].

All the above mentioned classes of attacks have in common that the Decode algorithm is not part of $\mathcal{A}$. Indeed, if we want to achieve non-malleability, then we must have that $\mathsf{Decode} \notin \mathcal{A}$, as otherwise the following attack becomes possible. Let A be the tampering algorithm that first decodes the codeword c to get the encoded value x, and then, depending on the first bit b of x, it overwrites c with c_b, where $\mathsf{Decode}(c_0) \neq \mathsf{Decode}(c_1)$. In this work, we aim at codes that achieve a weaker security guarantee than standard non-malleability, but for the first time can protect

[1] Sometimes, the tampering algorithms are also called tampering functions.

the security of cryptographic functionalities $\mathcal{F}$ with respect to a class of tampering attacks $\mathcal{A}$ with $\mathsf{Decode} \in \mathcal{A}$.

On the importance of $\mathsf{Decode} \in \mathcal{A}$. Besides being an obvious extension of the class of tampering attacks for which we can design non-malleable codes (albeit achieving a weaker security guarantee, which we will outline in Section 1.1), allowing that $\mathsf{Decode} \in \mathcal{A}$ has some important advantages for cryptographic applications, as emphasized by the following example. Consider a physical device storing an encoded key $\mathsf{Encode}(\kappa)$ in memory, and implementing a cryptographic functionality $\mathcal{F}$. If the device attempts to implement the cryptographic functionality $\mathcal{F}$, then whenever it is executed, it has to run the Decode function to recover the original secret key κ before running $\mathcal{F}(\kappa, \cdot)$. Suppose that a malicious piece of software A, e.g., a virus, infects the device and attempts to learn information about the secret key κ. Clearly, once A infects the device, it may use the resources available on the device itself, which in particular have to be sufficient to run the Decode algorithm. Hence, if we view the virus A as the tampering algorithm, to maintain the functionality of the device (which in particular requires to run Decode) and at the same time to allow the virus A to control the resources of attacked device, it is necessary that $\mathsf{Decode} \in \mathcal{A}$.[2] Our main contribution is to design non-malleable codes that can guarantee meaningful security in the above described setting. We provide more details on our results in the next section.

1.1 Our Contribution

Leaky non-malleable codes. The standard non-malleability property guarantees that decoding the tampered codeword reveals nothing about the original encoded message x. Formally, this is modelled by a simulation-based argument, where we consider the following tampering experiment. First, the message x gets encoded to $c \leftarrow \mathsf{Encode}(x)$ and the adversary can apply a tampering algorithm $\mathsf{A} \in \mathcal{A}$ resulting in a modified codeword $\tilde{c}$; the output of the tampering experiment is then defined as $\mathsf{Decode}(\tilde{c})$. Roughly speaking, non-malleability is guaranteed if we can construct an (efficient) simulator S that can produce a distribution that is (computationally) indistinguishable from the output of the tampering experiment, without having access to x; the simulator is typically allowed to return a special symbol $\mathsf{same}^\star$ to signal that (it believes) the adversarial tampering did not modify the encoded message.

As explained above, if $\mathsf{Decode} \in \mathcal{A}$, then the above notion is trivially impossible to achieve, since the adversary can easily learn $O(\log k)$ bits, where k is the size of the message.[3] In this work, we introduce a new notion that we call *leaky non-malleability*, which models the fact that, when $\mathsf{A} \in \mathcal{A}$, the adversary is allowed to learn some (bounded) amount of information about the message x. Formally, we give the simulator S additional access to a leakage oracle; more concretely, this means that in order to simulate the output of the tampering experiment, S can specify a leakage function $L : \{0,1\}^k \to \{0,1\}^\ell$ and receive $L(x)$.[4] Clearly, if $\ell = k$, then the simulation is trivial, and hence our aim is to design codes where ℓ is as close as possible to the necessary bound of $O(\log k)$. Notice that, due to the allowed leakage, our notion of leaky non-malleability makes most sense when the message x is sampled from a distribution of high min-entropy. But, indeed, this is the case in the main application of NMC, where the goal is to

[2] In particular, when resources are measured by space as considered in this work, assuming that running Decode requires more space than what is available on the device would imply assuming a trusted part of memory that the virus cannot exploit, which seems unnatural.

[3] For instance, the adversary may just guess the first $O(\log k)$ bits of the message and replace c with c_u (where $u \in \{0,1\}^{O(\log k)}$) depending on whether its guess was correct; this attack succeeds with non-negligible probability.

[4] Although, later in the paper, we define leaky non-malleability only for the case of space-bounded tampering, we point out that this weaker security guarantee makes sense for arbitrary tampering classes $\mathcal{A}$.

protect a secret key of a cryptographic scheme; and in fact, as we show at the end of the paper, leaky non-malleability still allows to guarantee protection against memory tampering in many interesting cases.

Modelling space-bounded tampering adversaries. In the above application with the virus, we allow the virus to use all resources of the device when it tampers with the codeword. Of course, this means that the virus is limited in the *amount of space* it can use. We exploit this observation by putting forward the notion of non-malleable codes that resist adversaries operating in bounded space. That is, in contrast to earlier works on NMC, we do not require any independence of the tampering (like, e.g., in the split-state model), nor the fact that tampering comes from a restricted complexity class. Instead, we allow arbitrary efficient tampering attacks that can globally modify the codeword, as long as the attacks operate in the space available on the device. Since the lower bounds in space complexity are notoriously hard, we follow earlier works [20, 19, 4, 18] that argue about space-bounded adversaries (albeit in a different setting), using the random oracle methodology and its connection to graph pebbling games.

Let us provide some more details on our model. Our setting follows the earlier work of Dziembowski, Kazana and Wichs [20, 19] and considers a "big adversary" B that has unlimited space (though runs in PPT) and creates "small adversaries" A (e.g., viruses) that it sends to the device. On the device, A can use the available space to modify the codeword in some arbitrary way. We emphasize that A has no granular restrictions, and hence can read the entire codeword. Moreover, it can follow an arbitrary efficient (PPT) tampering strategy. The only restriction is that A has to operate in bounded space. Both adversaries A and B have access to a random oracle $\mathcal{H}$. After A has finished its tampering attack, we proceed as in the normal NMC experiment, i.e., we decode the modified codeword and output the result. We further strengthen our definition by allowing the adversary to repeat the above attack multiple times, which is sometimes referred to as continuous tampering [25, 33]. We note that, as in [33], we require an a-priori fixed upper bound on the number of viruses A that B can adaptively choose.

Technical overview of our construction. Our construction is based on Proofs of Space (a.k.a. PoS), introduced in [4, 18]. First, let us recall the notion of PoS briefly. In a PoS protocol, a prover P proves that "it has sufficient space" available to a space-bounded verifier V. Using the Fiat-Shamir [29] transformation, the entire proof can be presented by π_{id} for some identity id. The verifier can verify the pair (id, π_{id}) within bounded space (say s). The soundness guarantee is that a cheating prover, with overwhelming probability, can not produce a correct proof unless it uses a large amount of space. Our NMC construction encodes a value $x \in \{0,1\}^k$ by setting $id := x$ and then computing the proof π_{id}. Hence, the codeword is $c = (x, \pi_x)$. Decoding is done just by running the verification procedure of the PoS.

Now, if the codeword is stored in an s-bounded device, then decoding is possible within the available space whereas encoding is not – in particular, even if the adversary can obtain x, it can not re-encode to a related value, say $(x+1, \pi_{x+1})$, as guaranteed by the soundness of the underlying PoS.[5] We stress that our soundness requirement is slightly different than the existing PoS constructions, as we require some form of "extractability" from the PoS: Given an honestly generated pair (x, π_x), if the space-bounded virus can compute a valid pair $(x', \pi_{x'})$ where $x' \neq x$, then one can efficiently extract x' from the set of random oracle queries that the big adversary made before installing the virus. Our put differently, the only way to compute a valid proof is to overwrite (x, π_x) with a valid pair $(x', \pi_{x'})$ "pre-computed" by the big adversary.

To formally prove the leaky non-malleability of our construction, we need to show that the

[5] Notice that since the space-bounded attacker A is able to decode anyway, we do not aim to hide x in c.

output of the tampering experiment can be simulated given only "limited" leakage on x. For simplicity, let us explain how this can be done for one tampering query. Intuitively this is possible because the big adversary can hard-code at most polynomially many (say q) correct pairs $\{x_i, \pi_{x_i}\}_{i \in [q]}$ into the virus. Now, since any such $x_i \neq x$ can be efficiently "extracted" from the random oracle queries made by B prior to choosing the virus, $\log(q)$ bits of leakage are sufficient to compute the exact x_i from the list $\{x_i\}_{i \in [q]}$.[6] For multiple adaptive tampering queries things get more complicated. Nonetheless, we are able to show that each such query can be simulated by logarithmic leakage.

We emphasize that our encoding scheme is deterministic for a fixed choice of the random oracle. In particular, the only randomness comes from the random oracle itself. Also, in the security proof, we do not require to program the random oracle in the on-line phase of the security reduction, in that the random oracle can just be fixed at the beginning of the security game.[7]. We concretely instantiate our construction by adapting the PoS protocol from Ren and Devadas [40], that uses so-called stacks of localized expander graphs.

Applications: Trading leakage for tamper resilience. One may ask if our notion of leaky non-malleability is useful for the original application of tamper protection. In Section 7 we show that cryptographic primitives which remain secure if the adversary obtains some bounded amount of leakage from the key, can naturally be protected against tampering attacks using our new notion of leaky non-malleability. Since there is a large body of work on bounded leakage-resilient cryptographic primitives, including signature schemes, symmetric and public key encryption [32, 16, 34, 38, 39, 22, 23], and many more, our transformation protects these primitives against any efficient space-bounded tampering attack.

1.2 Additional Related Work

Only very few works consider non-malleable codes for global tampering functions [5]. Very related to our attack model are in particular the works of Dziembowski, Kazana and Wichs [20, 19]. In these works, the authors also consider a setting where a so-called "big-adversary" infects a machine with a space-bounded "small adversary". Using techniques from graph pebbling, the authors show how to construct one-time computable functions [20] and leakage resilient key evolution schemes [19] when the "small adversary" has to operate in bounded space.

The flavor of non-malleable codes in which there is an a-priory upper bound on the total number of tampering queries, without self-destruct, was originally considered in [9]. This concept has a natural application to the setting of bounded tamper resilience (see, e.g., [15, 14, 24]).

For other related works on non-malleable codes and its applications we refer to [37].

2 Preliminaries

2.1 Notation

For a string x, we denote its length by $|x|$; if $\mathcal{X}$ is a set, $|\mathcal{X}|$ represents the number of elements in $\mathcal{X}$. When x is chosen randomly in $\mathcal{X}$, we write $x \leftarrow \mathcal{X}$. When A is an algorithm, we write $y \leftarrow \mathsf{A}(x)$ to denote a run of A on input x and output y; if A is probabilistic, then y is a random variable and $\mathsf{A}(x; r)$ denotes a run of A on input x and randomness r. An algorithm A

[6] In slightly more detail, the set $\{x_i\}_{i \in [q]}$ can be extracted by the simulator outside the leakage oracle as it does not depend on x, so the simulator can just ask for the index of the exact x_i to later reconstruct x_i in full.

[7] Since adaptive (i.e. on-line) programming is not required, for all practical purposes our construction can be instantiated by standard hash functions like SHA-1. However, our proof crucially relies on the ability of the simulator to control the random oracle (albeit non adaptively), in order to make the "extraction" work.

is *probabilistic polynomial-time* (PPT) if A is probabilistic and for any input $x, r \in \{0,1\}^*$ the computation of $\mathsf{A}(x;r)$ terminates in at most a polynomial (in the input size) number of steps. We often consider algorithms $\mathsf{A}^{\mathcal{O}(\cdot)}$, with access to an oracle $\mathcal{O}(\cdot)$.

We denote with $\lambda \in \mathbb{N}$ the security parameter. A function $\nu : \mathbb{N} \to [0,1]$ is negligible in the security parameter (or simply negligible), denoted $\nu(\lambda) \in \text{negl}(\lambda)$, if it vanishes faster than the inverse of any polynomial in λ, i.e. $\nu(\lambda) = \lambda^{-\omega(1)}$. A function $\mu : \mathbb{N} \to \mathbb{R}$ is a polynomial in the security parameter, written $\mu(\lambda) \in \text{poly}(\lambda)$, if, for an arbitrary constant $c > 0$, we have $\mu(\lambda) \in O(\lambda^c)$.

2.2 Coding Schemes

We recall the standard notion of a coding scheme for binary messages.

Definition 1 (Coding scheme)**.** A (k,n)-code $\Pi = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Decode})$ is a triple of algorithms specified as follows: (i) The (randomized) generation algorithm Init takes as input $\lambda \in \mathbb{N}$ and returns public parameters $\omega \in \{0,1\}^*$; (ii) The (randomized) encoding algorithm Encode takes as input hard-wired public parameters $\omega \in \{0,1\}^*$ and a value $x \in \{0,1\}^k$, and returns a codeword $c \in \{0,1\}^n$; (iii) The (deterministic) decoding algorithm Decode takes as input hard-wired public parameters $\omega \in \{0,1\}^*$ and a codeword $c \in \{0,1\}^n$, and outputs a value in $\{0,1\}^k \cup \{\perp\}$, where $\perp$ denotes an invalid codeword.

We say that Π satisfies correctness if for all $\omega \in \{0,1\}^*$ output by $\mathsf{Init}(1^\lambda)$ and for all $x \in \{0,1\}^k$, $\mathsf{Decode}_\omega(\mathsf{Encode}_\omega(x)) = x$ with overwhelming probability over the randomness of the encoding algorithm.

In this paper we will be interested in modelling coding schemes where there is an explicit bound on the space complexity required to decode a given codeword.

Definition 2 (Time/space-bounded algorithm)**.** Let A be an algorithm. For any $s, t \in \mathbb{N}$ we say that A is s-space bounded and t-time bounded (or simply (s,t)-bounded) if at any time during its execution the entire state of A can be described by at most s bits and A runs for at most t time-steps.

For such algorithms we have $s_{\mathsf{A}} \leq s$ and $t_{\mathsf{A}} \leq t$ (with the obvious meaning). We often omit the time parameter and simply say that A is s-bounded, which means that A is an s-bounded polynomial-time algorithm. Given an input $x \in \{0,1\}^n$, and an initial configuration $\sigma \in \{0,1\}^{s-n}$, we write $(y, \tilde{\sigma}) := \mathsf{A}(x;\sigma)$ for the output y of A including its final configuration $\tilde{\sigma} \in \{0,1\}^{s-n}$. The class of all s-space bounded deterministic polynomial-time algorithms is denoted by $\mathcal{A}^s_{\text{space}}$.

We stress that, similarly to previous works [20, 19], in case A is modelled as a Turing machine, we count the length of the input tape and the position of all the tape heads within the space bound s. However we emphasize that, although A is space-bounded, we allow to hard-wire auxiliary information of arbitrary polynomial length in its description that is not accounted for in the space-bound. Intuitively, a coding scheme can be decoded in bounded space if the decoding algorithm is space bounded.

Definition 3 (Space-bounded decoding)**.** Let $\Pi = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Decode})$ be a (k,n)-code, and $d \in \mathbb{N}$. We call Π a (k,n)-code with d-bounded decoding, if for all ω output by $\mathsf{Init}(1^\lambda)$ the decoding algorithm $\mathsf{Decode}_\omega(\cdot)$ is d-bounded.

Notice that we do not count the length of the public parameters in the space bound; this is because the value ω is hard-coded into the description of the encoding and decoding algorithms.

3 Non-Malleability in Bounded Space

In this paper we consider non-malleable codes against the class of tampering attacks that are modelled as s-bounded efficient algorithms, for some parameter $s \in \mathbb{N}$ (cf. Definition 2). Our model is motivated by the fact that a tampering attempt against a codeword, stored in some memory-constrained device, cannot use more space than the total amount of space available on the device itself. We define our model formally in Section 3.1, whereas in Section 3.2 we state simple bounds on the achievable range for the parameters in our definition.

3.1 Space-Bounded Tampering

The standard way of formalizing the non-malleability property is to require that, for any "allowed adversary"[8] A, tampering with an honestly computed target encoding of some value $x \in \{0,1\}^k$, there exists an efficient simulator S that is able to emulate the outcome of the decoding algorithm on the tampered codeword, without knowing x. The simulator is allowed to return a special symbol $\mathsf{same}^\star$, signalling that (it believes) the adversary did not modify the value x contained in the original encoding.

Below, we formalize non-malleability in the case where the set of allowed adversaries consists of all efficient s-bounded algorithms, for some parameter $s \in \mathbb{N}$ (cf. Definition 2). However, since we are particularly interested in decoding algorithms that are d-bounded for some value $d \leq s$, the standard notion of non-malleability is impossible to achieve, as in such a case the algorithm A can simply decode the tampered codeword and leak some information on the encoded message via tampering (see also the discussion in Section 3.2). To overcome this obstacle, we will give the simulator S some extra-power, in that S will additionally be allowed to obtain some limited amount of information on x in order to simulate the view of A. To capture this, we introduce an oracle $\mathcal{O}^{\ell,x}_{\text{leak}}$ that can be queried in order to retrieve up-to ℓ bits of information about x.

Definition 4 (Leakage oracle)**.** A *leakage oracle* $\mathcal{O}^{\ell,x}_{\text{leak}}$ is a stateful oracle that maintains a counter $\mathtt{ctr}$ that is initially set to 0. The oracle is parametrized by a string $x \in \{0,1\}^k$ and a value $\ell \in \mathbb{N}$. When $\mathcal{O}^{\ell,x}_{\text{leak}}$ is invoked on a polynomial-time computable leakage function L, the value $L(x)$ is computed, its length is added to $\mathtt{ctr}$, and if $\mathtt{ctr} \leq \ell$, then $L(x)$ is returned; otherwise, $\perp$ is returned.

Since our main construction is in the random oracle model (a.k.a. ROM), we will define space-bounded non-malleability explicitly for this setting. Recall that in the ROM a hash function $\mathcal{H}(\cdot)$ is modelled as an external oracle implementing a random function, which can be queried by all algorithms (including the adversary); in the simulation, the simulator S simulates the random oracle. We introduce the notion of a tampering oracle, which essentially corresponds to repeated (adaptive) tampering with a target n-bit codeword, using at most s bits of total space. Below, we consider that the total space of length s is split into two parts: (i) Persistent space of length p, that also stores the codeword of length n, and that is never erased by the oracle; and (ii) Transient (or non-persistent) space, of length $s - p$, that is erased by the oracle after every tampering. Looking ahead, in our tampering application (cf. Section 7), the persistent space corresponds to the user's hard-drive (storing arbitrary data), while the transient space corresponds to the transient memory available on the device.

Definition 5 (Space-bounded tampering oracle)**.** A *space-bounded tampering oracle* $\mathcal{O}^{\Pi,x,\omega,s,p}_{\text{cnm}}$ is a stateful oracle parameterized by a (k,n)-code $\Pi = (\mathsf{Init}^{\mathcal{H}}, \mathsf{Encode}^{\mathcal{H}}, \mathsf{Decode}^{\mathcal{H}})$, a string

[8] The adversary is often referred to as the "tampering function"; however, for our purposes, it is more convenient to think of the tampering function as an algorithm.

$x \in \{0,1\}^k$, public parameters $\omega \in \{0,1\}^*$, and values $s, p \in \mathbb{N}$ (with $s \geq p \geq n$). The oracle has an initial state $\texttt{st} := (c, \sigma)$, where $c \leftarrow \mathsf{Encode}_\omega^{\mathcal{H}}(x)$, and $\sigma := \sigma_0 || \sigma_1 := 0^{p-n} || 0^{s-p}$. Hence, upon input a deterministic algorithm $\mathsf{A} \in \mathcal{A}_{\text{space}}^s$, the output of the oracle is defined as follows.

```
Oracle $\mathcal{O}_{\text{cnm}}^{\Pi,x,\omega,s,p}(\mathsf{A})$:
Parse $\texttt{st} = (c, \sigma_0, \sigma_1)$
Let $(\tilde{c}, \tilde{\sigma}_0, \tilde{\sigma}_1) := \mathsf{A}^{\mathcal{H}}(c; \sigma_0 || \sigma_1)$
Return $\tilde{x} := \mathsf{Decode}_\omega^{\mathcal{H}}(\tilde{c})$
Update $\texttt{st} := (\tilde{c}, \tilde{\sigma}_0, 0^{s-p})$.
```

Notice that in the definition above we put space restrictions only on the tampering algorithm A. The oracle itself is space unbounded. In particular, this means that even if the decoding algorithm requires more space than s, the oracle is well defined. Moreover, this allows us to assume that the auxiliary persistent space $\tilde{\sigma}_0$ is never erased/overwritten by the oracle.

Furthermore, each algorithm A takes as input a codeword $\tilde{c}$ which is the result of the previous tampering attempt. In the literature, this setting is sometimes called persistent continuous tampering [33]. However, a closer look into our setting reveals that the model is actually quite different. Note that, the auxiliary persistent space σ_0 (that is the persistent space left after storing the codeword) can be used to copy parts of the original encoding, that thus can be mauled multiple times. (In fact, as we show in Section 3.2, if $p = 2n$, the above oracle actually allows for non-persistent tampering as considered in [33, 25].)

In the definition of non-malleability we will require that the output of the above tampering oracle can be simulated given only ℓ bits of leakage on the input x. We formalize this through a simulation oracle, which we define below.

Definition 6 (Simulation oracle)**.** A *simulation* oracle $\mathcal{O}_{\text{sim}}^{\mathsf{S}_2,\ell,x,s,\omega}$ is an oracle parametrized by a stateful PPT algorithm S_2, values $\ell, s \in \mathbb{N}$, some string $x \in \{0,1\}^k$, and public parameters $\omega \in \{0,1\}^*$. Upon input a deterministic algorithm $\mathsf{A} \in \mathcal{A}_{\text{space}}^s$, the output of the oracle is defined as follows.

```
Oracle $\mathcal{O}_{\text{sim}}^{\mathsf{S}_2,\ell,x,s,\omega}(\mathsf{A})$:
Let $\tilde{x} \leftarrow \mathsf{S}_2^{\mathcal{O}_{\text{leak}}^{\ell,x}(\cdot)}(1^\lambda, \omega, \mathsf{A})$
If $\tilde{x} = \mathsf{same}^\star$, set $\tilde{x} = x$
Return $\tilde{x}$.
```

We are now ready to define our main notion of continuous non-malleability under space-bounded tampering.

Definition 7 (Space-bounded continuous non-malleability)**.** Let $\mathcal{H}$ be a hash function modelled as a random oracle, and let $\Pi = (\mathsf{Init}^{\mathcal{H}}, \mathsf{Encode}^{\mathcal{H}}, \mathsf{Decode}^{\mathcal{H}})$ be a (k, n)-code. For parameters $\ell, s, p, \theta, d \in \mathbb{N}$, with $s \geq p \geq n$, we say that Π is an ℓ-leaky (s, p)-space-bounded θ-continuously non-malleable code with d-bounded decoding ((ℓ, s, p, θ, d)-SP-NMC for short) in the ROM, if it satisfies the following conditions.

- **Space-bounded decoding:** The decoding algorithm $\mathsf{Decode}^{\mathcal{H}}$ is d-bounded.
- **Non-malleability:** For all PPT distinguishers D, there exists a PPT simulator $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$ such that for all values $x \in \{0,1\}^k$ there is a negligible function $\nu : \mathbb{N} \to [0,1]$ satisfying

$$\Big| \Pr\left[\mathsf{D}^{\mathcal{H}(\cdot), \mathcal{O}_{\text{cnm}}^{\Pi,x,\omega,s,p}(\cdot)}(\omega) = 1 : \ \omega \leftarrow \mathsf{Init}^{\mathcal{H}}(1^\lambda)\right] \\ - \Pr\left[\mathsf{D}^{\mathsf{S}_1(\cdot), \mathcal{O}_{\text{sim}}^{\mathsf{S}_2,\ell,x,s,\omega}(\cdot)}(\omega) = 1 : \ \omega \leftarrow \mathsf{Init}^{\mathsf{S}_1}(1^\lambda)\right] \Big| \leq \nu(\lambda),$$

where D asks at most θ queries to $\mathcal{O}_{\text{cnm}}$. The probability is taken over the choice of the random oracle $\mathcal{H}$, the sampling of the initial state for the oracle $\mathcal{O}_{\text{cnm}}$, and the random coin tosses of D and $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$.

Intuitively, in the above definition algorithm S_1 takes care of simulating random oracle queries, whereas S_2 takes care of simulating the answer to tampering queries. Typically, S_1 and S_2 are allowed to share a state, but we do not explicitly write this for simplifying notation. For readers familiar with the notion of non-malleable codes in the common reference string model (see, e.g., [35, 25]), we note that the simulator is not required to program the public parameters (but is instead allowed to program the random oracle).[9]

Remark 1. *Note that we consider the space-bounded adversary* A *as deterministic; this is without loss of generality, as the distinguisher* D *can always hard-wire the "best randomness" directly into* A. *Also,* A *does not explicitly take the public parameters* ω *as input; this is also without loss of generality, as* D *can always hard-wire* ω *in the description of* A.

3.2 Achievable Parameters

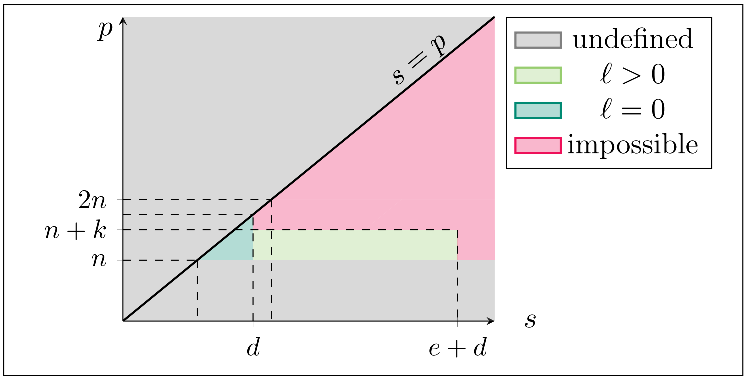

Figure 1: Possible values for the parameters $s, p \in \mathbb{N}$ in the definition of leaky space-bounded non-malleability, for fixed values of k, n, d (assuming $d < 2n$); in the picture, "impossible" means for $\theta \geq k$ and for non-trivial values of ℓ, and e is the space bound for the encoding algorithm.

We now make a few remarks on our definition of space-bounded non-malleability, and further investigate for which range of the parameters s (total space available for tampering), p (persistent space available for tampering), θ (number of adaptive tampering queries), d (space required for decoding), and ℓ (leakage bound), our notion is achievable. Let $\Pi = (\mathsf{Init}^{\mathcal{H}}, \mathsf{Encode}^{\mathcal{H}}, \mathsf{Decode}^{\mathcal{H}})$ be a (k, n)-code in the ROM.[10] First, note that leaky space-bounded non-malleability is trivial to achieve whenever $\ell = k$ (or $\ell = k - \varepsilon$, for $\varepsilon \in O(\log \lambda)$); this is because, for such values of the leakage bound, the simulator can simply obtain the input message $x \in \{0,1\}^k$, in which case the security guarantee becomes useless. Second, the larger the values of s and θ, the larger is the class of tampering attacks and the number of tampering attempts that the underlying code can tolerate. So, the challenge is to construct coding schemes tolerating a large space bound in the presence of "many" tampering attempts, using "small" leakage.

An important feature that will be useful for characterizing the range of achievable parameters in our definition is the so-called *self-destruct capability*, which determines the behavior of the decoding algorithm after an invalid codeword is ever processed. In particular, a code with the self-destruct capability is such that the decoding algorithm always outputs $\perp$ after the first $\perp$ is ever returned (i.e., after the first invalid codeword is ever decoded). Such a feature, which was already essential in previous works studying continuously non-malleable codes [25, 12, 11], can

[9] However, we stress that in the proof of our code construction (cf. Section 6), we do not need adaptive random oracle programming.

[10] The discussion below applies also to codes not relying on random oracles.

be engineered by enabling the decoding function to overwrite the entire memory content with a fixed string, say the all-zero string if a codeword is decoded to $\bot$.

Depending on the self-destruct capability being available or not, we have the following natural observations:

- If Π is not allowed to self-destruct, it is impossible to achieve space-bounded non-malleability, for non-trivial values of ℓ, whenever $\theta \geq n$ (for any $s \geq p \geq n$, and any $d \in \mathbb{N}$). This can be seen by considering the deterministic algorithm $\mathsf{A}^i_{\mathsf{aux}_i}$ (for some $i \in [n]$) that overwrites the first $i-1$ bits of the input codeword with the values $\mathsf{aux}_i := (c[1], \ldots, c[i-1])$, and additionally sets the i-th bit to 0 (leaving the other bits unchanged). Using such an algorithm, a PPT distinguisher D can guess the bit $c[i]$ of the target codeword to be either 0 (in case the tampering oracle returned the input message x) or 1 (in case the tampering oracle returned a value different from x, namely $\bot$). Hence, D returns 1 if and only if $\mathsf{Decode}^{\mathcal{H}}_{\omega}(c) = x$.

  The same attack was already formally analyzed in [12] (generalizing a previous attack by Gennaro *et al.* [31]); it suffices to note here that the above attack can be mounted using $s = n$ bits of space (which are needed for processing the input encoding), and requires $\theta = n$ tampering attempts.

- Even if Π is allowed to self-destruct, whenever $s \geq d$ and $p \geq n+\theta-1$, leaky space-bounded non-malleability requires $\ell \geq \theta$. This can be seen by considering the following attack. An s-bounded algorithm $\mathsf{A}^1_{c_0,c_1}$, with hard-wired two valid encodings $c_0, c_1 \in \{0,1\}^n$ of two *distinct* messages $x_0, x_1 \in \{0,1\}^k$ does the following: (i) Decodes c obtaining x (which requires $d \leq s$ bits of space); (ii) Stores the first $\theta - 1$ bits of x in the persistent storage $\tilde{\sigma}_0$; (iii) If the θ-th bit of x is one, it replaces c with $\tilde{c} = c_1$, else it replaces c with $\tilde{c} = c_0$. During the next tampering query, D can specify an algorithm $\mathsf{A}^2_{c_0,c_1}$ that overwrites the target encoding with either c_0 or c_1 depending on the first[11] bit of $\tilde{\sigma}_0$ being zero or one, and so on until the first $\theta - 1$ bits of x are leaked. So in total, it is able to leak at least θ bits of x (including the θ-th bit of x leaked by A^1).

- The previous attack clearly implies that it is impossible to achieve leaky space-bounded non-malleability, for non-trivial values of ℓ, whenever $s \geq d$, $\theta = k$, and $p \geq n + k - \varepsilon$, for $\varepsilon \in O(\log \lambda)$. A simple variant of the above attack, where essentially D aims at leaking the target encoding c instead of the input x, yields a similar impossibility result whenever $s \geq p$, $d \in \mathbb{N}$, $\theta = n$, and $p \geq 2n - \varepsilon$, for $\varepsilon \in O(\log \lambda)$.

The above discussion is summarized in the following theorem (see also Fig. 1 for a pictorial representation).

Theorem 1. *Let $\ell, s, p, \theta, d, k, n \in \mathbb{N}$ be functions of the security parameter $\lambda \in \mathbb{N}$. The following holds:*

(i) No (k,n)-code Π without the self-destruct capability can be an (ℓ, s, p, θ, d)-SP-NMC for $d \in \mathbb{N}$, $s \geq p \geq n$ and $\ell = n - \mu$, where $\mu \in \omega(\log \lambda)$.

(ii) For any $1 \leq \theta < k$, if Π is a (k,n)-code (with or without the self-destruct capability) that is an (ℓ, s, p, θ, d)-SP-NMC for $d \in \mathbb{N}$, $s \geq d$ and $p \geq n + \theta - 1$, then $\ell \geq \theta$.

[11] Recall that the tampering oracle of Definition 5 initializes the persistent space σ_0 used by the current tampering algorithm, with the corresponding final state $\tilde{\sigma}_0$ returned by the previous tampering algorithm.

(iii) No (k,n)-code Π (even with the self-destruct capability) can be an (ℓ, s, p, θ, d)-SP-NMC for $d \in \mathbb{N}$, $\ell = n - \mu$, with $\mu \in \omega(\log \lambda)$, where, for $\varepsilon \in O(\log \lambda)$,

$$\begin{array}{rcc} s \geq d & \theta \geq k & p \geq n + k - \varepsilon \\ \text{or } s \geq p & \theta \geq n & p \geq 2n - \varepsilon. \end{array}$$

Remark 2. *We emphasize that our coding scheme (cf. Section 6) does* not *rely on any self-destruct mechanism, and achieves $\theta \approx k/\log \lambda$ for non-trivial values of the leakage parameter. This leaves open the question to construct a code relying on the self-destruct capability, that achieves security for any $\theta \in \text{poly}(\lambda)$ and for non-trivial leakage, with parameters s, p, d consistent with the above theorem. We leave this as an interesting direction for future research.*

4 Building Blocks

In this section we define the main cryptographic primitives on which we base our construction later. We start by putting forward a few conventions about random oracles, in Section 4.1, and by recalling the standard properties of Merkle trees, in Section 4.2. Then, in Section 4.3, we define a few concepts related to random-oracle labeling of directed acyclic graphs.

4.1 Random Oracles

All our results are in the random oracle model (ROM). Therefore we first discuss some basic conventions and definitions related to random oracles. First, recall that in the ROM, at setup, a hash function $\mathcal{H}$ is sampled uniformly at random, and all algorithms, including the adversary, are given oracle access to $\mathcal{H}$ (unless stated otherwise). For instance, we let $\Pi = (\mathsf{Init}^{\mathcal{H}}, \mathsf{Encode}^{\mathcal{H}}, \mathsf{Decode}^{\mathcal{H}})$ be a coding scheme in the ROM. Second, without loss of generality, we will always consider a random oracle $\mathcal{H}$ with a type $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{n_{\mathcal{H}}}$.

We emphasize that unlike many other proofs in the ROM, we will not need the *full programmability* of random oracles. In fact, looking ahead, in the security proof of our code construction from Section 6, we can just assume that the random oracle is *non-adaptively programmable* as defined in [6].[12] The basic idea is that the simulator/reduction samples a partially defined "random-looking function" at the beginning of the security game, and uses that function as the random oracle $\mathcal{H}$. In particular, by fixing a function ahead of time, the reduction fixes all future responses to random oracle calls—this is in contrast to programmable random oracles, which allow the simulator to choose random values adaptively in the game, and also to program the output of the oracle in a convenient manner.

For any string x, and any random oracle $\mathcal{H}$, we use the notation $\mathcal{H}_x$ to denote the specialized random oracle that accepts only inputs with prefix equal to x. We additionally make the following conventions:

- **Query Tables.** Random oracle queries are stored in query tables. Let $\mathcal{Q}_{\mathcal{H}}$ be such a table. $\mathcal{Q}_{\mathcal{H}}$ is initialized as $\mathcal{Q}_{\mathcal{H}} := \emptyset$. Hence, when $\mathcal{H}$ is queried on a value u, a new tuple $(I(u), u, \mathcal{H}(u))$ is appended to the table $\mathcal{Q}_{\mathcal{H}}$ where $I : \{0,1\}^* \to \{0,1\}^{O(\log \lambda)}$ is an injective function that maps each input u to a unique identifier, represented in bits. Clearly, for any tuple $(i, u, \mathcal{H}(u))$ we have that $I^{-1}(i) = u$.

- **Input Field.** Let $\mathcal{Q}_{\mathcal{H}} = ((i_1, u_1, v_1), \cdots, (i_q, u_q, v_q))$ be a query table. The input field $\mathcal{IP}_{\mathcal{Q}_{\mathcal{H}}}$ of $\mathcal{Q}_{\mathcal{H}}$ is defined as the tuple $\mathcal{IP}_{\mathcal{Q}_{\mathcal{H}}} = (u_1, \ldots, u_q)$.

[12] In [6], the authors show that such random oracles are equivalent to non-programmable ones, as defined in [30].

4.2 Merkle Commitments

Merkle Commitment

```
$\mathsf{MGen}^{\mathcal{H}}(1^\lambda)$: Output $\omega_{\mathsf{cm}} := \emptyset$.

$\mathsf{MCommit}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(z_0, \dots, z_{N-1})$: Output $\psi := \mathsf{Root}^{\mathcal{H}}(N, (z_0, \dots, z_{N-1}))$.

$\mathsf{MOpen}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}((z_0, \dots, z_{N-1}), i)$:
    • If $i \equiv 0 \mod 2$ then $\phi_1 := z_{i+1}$; else $\phi_1 := z_{i-1}$.
    • For $j = 2$ to $\log(N)$ do
        – $i := i$ div 2
        – If $i \equiv 0 \mod 2$ then $\phi_j := \mathsf{Root}^{\mathcal{H}}(2^{j-1}, (z_{(i+1)2^{j-1}}, \dots, z_{(i+2)2^{j-1}-1}))$;
          else $\phi_j := \mathsf{Root}^{\mathcal{H}}(2^{j-1}, (z_{(i-1)2^{j-1}}, \dots, z_{i2^{j-1}-1}))$.
    • Output $(z_i, (\phi_1, \dots, \phi_{\log N}))$.

$\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(i, \psi, (z, (\phi_1, \dots, \phi_{\log N})))$:
    • If $i \equiv 0 \mod 2$ then $\psi' := \mathcal{H}(z||\phi_1)$; else $\psi' := \mathcal{H}(\phi_1||z)$.
    • For $j = 2$ to $\log(N)$ do
        – $i := i$ div 2
        – If $i \equiv 0 \mod 2$ then $\psi' := \mathcal{H}(\psi'||\phi_j)$; else $\psi' := \mathcal{H}(\phi_j||\psi')$.
    • If $\psi = \psi'$ then output 1, otherwise output 0.
```

Figure 2: Construction of a Merkle commitment scheme

A Merkle commitment is a special type of commitment scheme[13] exploiting so-called hash trees [36]. Intuitively, a Merkle commitment allows a sender to commit to a vector of N elements $\mathbf{z} := (z_1, \dots, z_N)$ using $N-1$ invocations of a hash function. At a later point, the sender can open any of the values z_i, by providing a succinct certificate of size logarithmic in N.

Definition 8 (Merkle commitment)**.** A $(k, n_{\mathsf{cm}}, N, n_{\mathsf{op}}, \nu_{\mathsf{mt}})$-Merkle commitment scheme (or MC scheme) in the ROM is a tuple of algorithms $(\mathsf{MGen}^{\mathcal{H}}, \mathsf{MCommit}^{\mathcal{H}}, \mathsf{MOpen}^{\mathcal{H}}, \mathsf{MVer}^{\mathcal{H}})$ described as follows.

- $\mathsf{MGen}^{\mathcal{H}}(1^\lambda)$: On input the security parameter, the randomized algorithm outputs public parameters $\omega_{\mathsf{cm}} \in \{0,1\}^*$.
- $\mathsf{MCommit}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z})$: On input the public parameters and an N-tuple $\mathbf{z} = (z_1, \dots, z_N)$, where $z_i \in \{0,1\}^k$, this algorithm outputs a commitment $\psi \in \{0,1\}^{n_{\mathsf{cm}}}$.
- $\mathsf{MOpen}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}, i)$: On input the public parameters, a vector $\mathbf{z} = (z_1, \dots, z_N) \in \{0,1\}^{kN}$, and $i \in [N]$, this algorithm outputs an opening $(z_i, \phi) \in \{0,1\}^{n_{\mathsf{op}}}$.
- $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(i, \psi, (z, \phi))$: On input the public parameters, an index $i \in [N]$, and a commitment/opening pair $(\psi, (z, \phi))$, this algorithm outputs a decision bit.

[13] Commitment schemes typically also have *hiding*, which ensures that the commitment does not reveal any information about the committed string. Looking ahead, we will commit to a public string and hence hiding is not needed in our case.

We require the following properties to hold.

Correctness: For all $\mathbf{z} = (z_1, \dots, z_N) \in \{0,1\}^{kN}$, and all $i \in [N]$, we have that

$$\Pr\left[\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(i, \psi, (z_i, \phi)) = 1 : \begin{array}{r} \omega_{\mathsf{cm}} \leftarrow \mathsf{MGen}^{\mathcal{H}}(1^\lambda); \\ \psi \leftarrow \mathsf{MCommit}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}) \\ (z_i, \phi) \leftarrow \mathsf{MOpen}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}, i) \end{array}\right] = 1$$

Binding: For all $\mathbf{z} = (z_1, \dots, z_N) \in \{0,1\}^{kN}$, for all $i \in [N]$, and all PPT adversaries A, we have $\Pr[\mathbf{G}^{\mathsf{bind}}_{\mathsf{A},\mathbf{z},i}(\lambda) = 1] \le \nu_{\mathsf{mt}}$, where the game $\mathbf{G}^{\mathsf{bind}}_{\mathsf{A},\mathbf{z},i}(\lambda)$ is defined as follows:

Game $\mathbf{G}^{\mathsf{bind}}_{\mathsf{A},\mathbf{z},i}$:

1. Sample $\omega_{\mathsf{cm}} \leftarrow \mathsf{MGen}^{\mathcal{H}}(1^\lambda)$.
2. Let $(\psi, (z', \phi')) \leftarrow \mathsf{A}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}, i)$.
3. Let $(z_i, \phi_i) := \mathsf{MOpen}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}, i)$.
4. Output 1 if and only if all of the following conditions are satisfied:
   (a) $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(i, \psi, (z', \phi')) = 1$.
   (b) $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(i, \psi, (z_i, \phi_i)) = 1$.
   (c) $z' \neq z_i$.

Merkle commitments are a well known cryptographic primitive. For completeness, we recall how the construction works. Although such commitments can be based on standard collision-resistant hash functions, for consistency with the rest of the paper we describe the construction in the ROM.

Let T be a complete binary tree with N leafs (without lost of generality we assume that N is a power of two), and let $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^k$ be a random oracle. We assign an index $i \in [0, N-1]$ to every leaf of the tree T. Informally, to commit to a string $\mathbf{z}$, one first parses $(z_0, \dots, z_{N-1}) := \mathbf{z}$ and defines z_i to be the label of the i-th leaf of T. The commitment is then defined as the label of the root of the hash tree T. A more formal description is given in Fig. 2 (see also Fig. 3 for a pictorial representation). To simplify the notation in the figure, we define an auxiliary algorithm Root which, given the values of L leafs, recursively computes the value of the root of the binary hash tree. More formally:

Algorithm $\mathsf{Root}^{\mathcal{H}}(L, (z_0, \dots, z_{L-1}))$:

- If $L = 2$, then output $\mathcal{H}(z_0 || z_1)$;
- Else, output $\mathcal{H}\left(\mathsf{Root}^{\mathcal{H}}(\frac{L}{2}, (z_0, \dots, z_{\frac{L}{2}-1})) || \mathsf{Root}^{\mathcal{H}}(\frac{L}{2}, (z_{\frac{L}{2}}, \dots, z_{L-1}))\right)$.

4.3 Graph Pebbling and Labeling

Throughout this paper $G = (V, E)$ is considered to be a directed acyclic graph (DAG), where V is the set of vertices and E is the set of edges of the graph G. Without loss of generality we assume that the vertices of G are ordered lexicographically and are represented by integers in $[N]$, where $N = |V|$. Vertices with no incoming edges are called *input vertices* or *sources*, and vertices with no outgoing edges are called *output vertices* or *sinks*. We denote $\Gamma^-(v)$, the set of all predecessors of the vertex v. Formally, $\Gamma^-(v) = \{w \in V : (w, v) \in E\}$.

In this section we briefly explain the concept of graph labeling and its connection to the abstract game called graph pebbling which has been introduced in [17]. For more details we refer to previous literature in, e.g., [17, 40, 4, 18]. We follow conventions from [40] and will use results from the same. Sometimes for completeness we will use texts verbatim from the same paper.

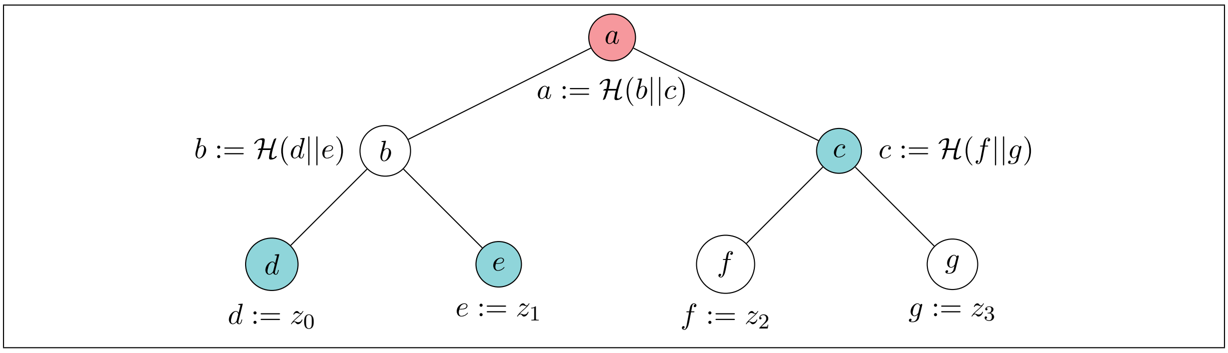

Figure 3: Example of a Merkle commitment of a string $\mathbf{z} = (z_0, z_1, z_2, z_3)$. Here $\mathsf{MCommit}^{\mathcal{H}}(\mathbf{z}) = a$ and for example $\mathsf{MOpen}^{\mathcal{H}}(1, \mathbf{z}) = (e, (d, c))$. The verification algorithm, given $(a, (e', (d', c')))$ and the index 1 computes $b' := \mathcal{H}(d'||e')$ and $a' := \mathcal{H}(b'||c')$ and outputs 1 if $a = a'$ and 0 otherwise.

Labeling of a graph. Let $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^{n_\mathcal{H}}$ be a random oracle. The $\mathcal{H}$-labeling of a graph G is a function which assigns a label to each vertex in the graph; more precisely, it is a function $\mathsf{label}\colon V \to \{0,1\}^{n_\mathcal{H}}$ which maps each vertex $v \in V$ to a bit string $\mathsf{label}(v) := \mathcal{H}(q_v)$, where we denote by $\left\{v^{(1)}, \dots, v^{(d)}\right\} = \Gamma^-(v)$ and let

$$q_v := \begin{cases} v & \text{if } v \text{ is an input vertex,} \\ v \mid\mid \mathsf{label}(v^{(1)}) \mid\mid \dots \mid\mid \mathsf{label}(v^{(d)}) & \text{otherwise.} \end{cases}$$

An algorithm $\mathsf{A}^\mathcal{H}$ labels a subset of vertices $W \subseteq V$ if it computes $\mathsf{label}(W)$. Specifically, $\mathsf{A}^\mathcal{H}$ labels the graph G if it computes $\mathsf{label}(V)$.

Additionally, for $m \leq |V|$, we define the $\mathcal{H}$-labeling of the graph G with m faults[14] as a function $\mathsf{label}\colon V \to \{0,1\}^{n_\mathcal{H}}$ such that, for some subset of vertices $M \subset V$ of size m, it holds

$$\begin{aligned} \mathsf{label}(v) &= \mathcal{H}(q_v), \text{ for every } v \in V \setminus M, \\ \mathsf{label}(v) &\neq \mathcal{H}(q_v), \text{ for every } v \in M. \end{aligned}$$

Sometimes we refer to labeling with faults as partial labeling. The following lemma appeared in form of a discussion in [40]. It is based on an observation previously made in [18].

Lemma 1 ([40, Section 5.2]). *Let $\mathsf{A}^\mathcal{H}$ be an (s,t)-bounded algorithm which computes the labeling of a DAG G with $m \in \mathbb{N}$ faults. Then there exists an $(s + m \cdot n_\mathcal{H}, t)$-bounded algorithm $\tilde{\mathsf{A}}^\mathcal{H}$ that computes the labeling of G without faults but gets m correct labels to start with (they are initially stored in the memory of $\tilde{\mathsf{A}}^\mathcal{H}$ and sometimes called initial labels).*

Intuitively the above lemma follows because the algorithm $\tilde{\mathsf{A}}^\mathcal{H}$ can overwrite the additional space it has, once the initial labels stored there are not needed.

Pebbling game. The pebbling of a DAG $G = (V, E)$ is defined as a single-player game. The game is described by a sequence of pebbling configurations $\mathbf{P} = (P_0, \dots, P_T)$, where $P_i \subseteq V$ is the set of pebbled vertices after the i-th move. In our model, the initial configuration P_0 does not need to be empty. The rules of the pebbling game are the following. During one move (translation from P_i to P_{i+1}), the player can place one pebble on a vertex v if v is an input vertex or if all predecessors of v already have a pebble. After placing one pebble, the player can

[14] One can also define an analogy of faults in the pebbling game by adding a second kind of pebbles. These pebbles are called *red pebbles* in [18] and *wild cards* in [4].

remove pebbles from arbitrary many vertices.[15] We say that the sequence $\mathbf{P}$ pebbles a set of vertices $W \subseteq V$ if $W \subseteq \bigcup_{i\in[0,T]} P_i$.

The time complexity of the pebbling game $\mathbf{P}$ is defined as the number of moves $t(\mathbf{P}) := T$. The space complexity of $\mathbf{P}$ is defined as the maximal number of pebbles needed at any pebbling step; formally, $s(\mathbf{P}) := \max_{i\in[0,T]}\{|P_i|\}$.

Ex-post-facto pebbling. Let $\mathsf{A}^{\mathcal{H}}$ be an algorithm that computes the (partial) $\mathcal{H}$-labeling of a DAG G. The ex-post-facto pebbling bases on the transcript of the graph labeling. It processes all oracle queries made by $\mathsf{A}^{\mathcal{H}}$ during the graph labeling (one at a time and in the order they were made). Informally, every oracle query of the form q_v, for some $v \in V$, results in placing a pebble on the vertex v in the ex-post-facto pebbling game. This provides us a link between labeling and pebbling of the graph G. The formal definition follows.

Let $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^{n_{\mathcal{H}}}$ be a random oracle and $\mathcal{Q}_{\mathcal{H}}$ a table of all random oracle calls made by $\mathsf{A}^{\mathcal{H}}$ during the graph labeling. Then we define the *ex-post-facto pebbling* $\mathbf{P}$ *of the graph* G as follows:

- The initial configuration P_0 contains every vertex $v \in V$ such that $\mathsf{label}(v)$ has been used for some oracle query (e.g. some query of the form $\mathcal{H}(\cdots \| \mathsf{label}(v) \| \cdots)$) at some point in the transcript but the query q_v is not listed in the part of the transcript preceding such query.

- Assume that the current configuration is P_i, for some $i \geq 0$. Then find the next unprocessed oracle query which is of the form q_v, for some vertex v, and define P_{i+1} as follows:

  1. Place a pebble on the vertex v.
  2. Remove all *unnecessary* pebbles. A pebble on a vertex v is called unnecessary if $\mathsf{label}(v)$ is not used for any future oracle query, or if the query q_v is listed in the succeeding part of the transcript before $\mathsf{label}(v)$ is used in an argument of some other query later. Intuitively, either $\mathsf{label}(v)$ is never used again, or $\mathsf{A}^{\mathcal{H}}$ anyway queries q_v before it is used again.

The lemma below appeared in several variations in the literature (see, for example, [17, 4, 40]), depending on the definition of graph pebbling.

Lemma 2 (Labeling Lemma). *Let G be a DAG. Consider an (s,t)-bounded adversary $\mathsf{A}^{\mathcal{H}}$ which computes the $\mathcal{H}$-labeling of the graph G. Also assume that $\mathsf{A}^{\mathcal{H}}$ does not guess any correct output of $\mathcal{H}$ without querying it. Then the ex-post facto pebbling strategy $\mathbf{P}$ described above pebbles the graph G, and the complexity of $\mathbf{P}$ is*

$$s(\mathbf{P}) \leq \frac{s}{n_{\mathcal{H}}} \quad \textit{and} \quad t(\mathbf{P}) \leq t.$$

Proof. By definition of ex-post-facto pebbling, it is straightforward to observe that if $\mathsf{A}^{\mathcal{H}}$ computes the $\mathcal{H}$-labeling of the graph G, then the ex-post-facto pebbling $\mathbf{P}$ pebbles the graph. Since we assume that the adversary does not guess the correct label, the only way $\mathsf{A}^{\mathcal{H}}$ can learn the label of the vertex v is by querying the random oracle. The bound on $t(\mathbf{P})$ is immediate. Again, by definition of the ex-post-facto pebbling, there is no unnecessary pebble at any time. Thus, the number of required pebbles is equal to the maximum number of labels that $\mathsf{A}^{\mathcal{H}}$ needs to store at once. Hence, the space bound follows directly from the fact that each label consists of $n_{\mathcal{H}}$ bits and that the algorithm $\mathsf{A}^{\mathcal{H}}$ is s-space bounded. □

[15] Similar to [40] in our model we assume that removing pebbles is for free as it does not involve any oracle query

Localized expander graphs. A (μ, α, β)-bipartite expander, for $0 < \alpha < \beta < 1$, is a DAG with μ sources and μ sinks such that any $\alpha\mu$ sinks are connected to at least $\beta\mu$ sources. We can define a DAG $G'_{\mu,k_G,\alpha,\beta}$ by stacking k_G ($\in \mathbb{N}$) bipartite expanders. Informally, stacking means that sinks of the i-th bipartite expander are sources of the i+1-st bipartite expander. It is easy to see that such a graph has $\mu(k_G + 1)$ nodes which are partitioned into $k_G + 1$ sets (which we call layers) of size μ. A Stack of Localized Expander Graphs (SoLEG) is a DAG $G_{\mu,k_G,\alpha,\beta}$ obtained by applying the transformation called *localization* (see [7, 40] for a definition) on each layer of the graph $G'_{\mu,k_G,\alpha,\beta}$.

We restate two lemmas about pebbling complexity of SoLEG from [40]. The latter appeared in [40] in form of a discussion.

Lemma 3 ([40, Theorem 4]). *Let $G_{\mu,k_G,\alpha,\beta}$ be a SoLEG where $\gamma := \beta - 2\alpha > 0$. Let $\mathbf{P} = (P_0, \ldots, P_{t(\mathbf{P})})$ be a pebbling strategy that pebbles at least $\alpha\mu$ output vertices of the graph $G_{\mu,k_G,\alpha,\beta}$ which were not initially pebbled, where the initial pebbling configuration is such that $|P_0| \leq \gamma\mu$, and the space complexity of $\mathbf{P}$ is bounded by $s(\mathbf{P}) \leq \gamma\mu$. Then the time complexity of $\mathbf{P}$ has the following lower bound:*

$$t(\mathbf{P}) \geq 2^{k_G}\alpha\mu.$$

Lemma 4 ([40, Section 5.2]). *Let $G_{\mu,k_G,\alpha,\beta}$ be a SoLEG and $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^{n_\mathcal{H}}$ be a random oracle. There exists a polynomial time algorithm $\mathsf{A}^\mathcal{H}$ that computes the $\mathcal{H}$-labeling of the graph $G_{\mu,k_G,\alpha,\beta}$ in $\mu n_\mathcal{H}$-space.*

5 Non-Interactive Proofs of Space

We start by defining non-interactive proofs of space, in Section 5.1. Then, in Section 5.2, we describe our construction based on Merkle commitments and pebbling of directed acyclic graphs. The security analysis is given in Section 5.3.

5.1 NIPoS Definition

A proof of space (PoS) [4, 18] is a (possibly interactive) protocol between a prover and a verifier, in which the prover attempts to convince the verifier that it used a considerable amount of memory or disk space in a way that can be easily checked by the verifier. Here, "easily" means with a small amount of space and computation; a PoS with these characteristics is sometimes called a proof of transient space [40]. A non-interactive PoS (NIPoS) is simply a PoS where the proof consists of a single message, sent by the prover to the verifier; to each proof, it is possible to associate an identity.

Intuitively, a NIPoS should meet two properties known as completeness and soundness. Completeness says that a prover using a sufficient amount of space will always be accepted by the verifier. Soundness, on the other hand, ensures that if the prover invests too little space, it has a hard time to convince the verifier. A formal definition follows below.

Definition 9 (Non-interactive proof of space). For parameters $s_\mathsf{P}, s_\mathsf{V}, s, t, k, n \in \mathbb{N}$, with $s_\mathsf{V} \leq s < s_\mathsf{P}$, and $\nu_\mathsf{pos} \in (0, 1)$, an $(s_\mathsf{P}, s_\mathsf{V}, s, t, k, n, \nu_\mathsf{pos})$-non-interactive proof of space scheme (NIPoS for short) in the ROM consists of a tuple of PPT algorithms $(\mathsf{Setup}^\mathcal{H}, \mathsf{P}^\mathcal{H}, \mathsf{V}^\mathcal{H})$ with the following syntax.

- $\mathsf{Setup}^\mathcal{H}(1^\lambda)$: This is a randomized polynomial-time (in λ) algorithm with no space restriction. It takes as input the security parameter and it outputs public parameters $\omega_\mathsf{pos} \in \{0,1\}^*$.

- $\mathsf{P}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id)$: This is a probabilistic polynomial-time (in λ) algorithm that is s_{P}-bounded. It takes as input an identity $id \in \{0,1\}^k$ and hard-wired public parameters ω_{pos}, and it returns a proof of space $\pi \in \{0,1\}^n$.
- $\mathsf{V}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id, \pi)$: This algorithm is s_{V}-bounded and deterministic. It takes as input an identity id, hard-wired public parameters ω_{pos}, and a candidate proof of space π, and it returns a decision bit.

We require the following properties to hold.

Completeness: For all $id \in \{0,1\}^k$, we have that

$$\Pr\left[\mathsf{V}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id, \pi) = 1 : \ \omega_{\mathsf{pos}} \leftarrow \mathsf{Setup}^{\mathcal{H}}(1^\lambda); \pi \leftarrow \mathsf{P}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id)\right] = 1,$$

where the probability is taken over the randomness of algorithms Setup and P, and over the choice of the random oracle.

Extractability: There exists a polynomial-time deterministic algorithm K (the knowledge extractor) such that for any probabilistic polynomial-time algorithm B, and for any $id \in \{0,1\}^k$, we have

$$\Pr[\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda) = 1] \leq \nu_{\mathsf{pos}},$$

where the experiment $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda)$ is defined as follows:

Game $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda)$:

1. Sample $\omega_{\mathsf{pos}} \leftarrow \mathsf{Setup}^{\mathcal{H}}(1^\lambda)$ and $\pi \leftarrow \mathsf{P}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id)$.
2. Let $\mathsf{A} \leftarrow \mathsf{B}^{\mathcal{H}}(\omega_{\mathsf{pos}}, id, \pi)$ and $\{id_i\}_{i\in[q]} := \mathsf{K}(\omega_{\mathsf{pos}}, \mathcal{Q}_{\mathcal{H}}(\mathsf{B}))$.
3. Let $(\tilde{id}, \tilde{\pi}) := \mathsf{A}^{\mathcal{H}}(id, \pi)$.
4. Output 1 if and only if $\mathsf{V}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(\tilde{id}, \tilde{\pi}) = 1$ and $\tilde{id} \notin \{id_i\}_{i\in[q]} \cup \{id\}$

where A is an (s,t)-bounded deterministic algorithm, $q \in \text{poly}(\lambda)$, the set $\mathcal{Q}_{\mathcal{H}}(\mathsf{B})$ contains the sequence of queries of B to $\mathcal{H}$ and the corresponding answers, and where the probability is taken over the coin tosses of $\mathsf{Setup}, \mathsf{B}, \mathsf{P}$, and over the choice of the random oracle.

Roughly, the extractability property requires that no space-bounded adversary is able to modify an honestly computed proof π for identity id into an accepting proof $\tilde{\pi}$ for an identity $\tilde{id} \neq id$. Moreover, this holds true even if A is chosen adaptively (possibly depending on the public parameters, the identity id, and a corresponding valid proof π) by a PPT algorithm B with unbounded space. Since, however, B can compute offline an arbitrary polynomial number of valid proofs (id_i, π_i), what the definition requires is that no (B, A) is able to yield a valid pair $(\tilde{id}, \tilde{\pi})$ for an $\tilde{id}$ different than id that the knowledge extractor K cannot predict by just looking at B's random oracle queries. It is easy to see that such an extractability requirement constitutes a stronger form of soundness, as defined, e.g., in [4, 40].

5.2 NIPoS Construction

We now give a NIPoS construction that is essentially a non-interactive variant of the PoS constructions of [40] that is in turn based on [4]. In particular, we show that it satisfies the stronger form of soundness which we call extractability. In addition, we formalize the security analysis given in [40] with concrete parameters that may be of independent interest.

The construction is built from the following ingredients:

- A random oracle $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{n_\mathcal{H}}$.
- A graph $G_{\mu,k_G,\alpha,\beta}$ from the family of SoLEG (cf. Section 4.3), where α, β are constants in $(0,1)$ such that $2\alpha < \beta$. By definition of such a graph, the number of nodes is given by $N = \mu(k_G + 1)$. The in-degree d depends on $\gamma = \beta - 2\alpha$, and it is hence constant.[16]

  Without loss of generality we assume that the vertices of $G_{\mu,k_G,\alpha,\beta}$ are ordered lexicographically and are represented by integers in $[N]$. For simplicity we also assume that N is a power of 2, and that $\log(N)$ divides $n_\mathcal{H}$.
- A $(n_\mathcal{H}, n_\mathsf{cm}, N, n_\mathsf{op}, \nu_\mathsf{mt})$-Merkle commitment scheme $(\mathsf{MGen}^\mathcal{H}, \mathsf{MCommit}^\mathcal{H}, \mathsf{MOpen}^\mathcal{H}, \mathsf{MVer}^\mathcal{H})$ (cf. Section 4.2).

Our construction is formally described in Fig. 4. Let us here just briefly explain the main ideas. The setup algorithm chooses a graph $G_{\mu,k_G,\alpha,\beta}$ from the family of SoLEG. Given an identity id, the prover first computes the $\mathcal{H}_{id}$-labeling of the graph $G_{\mu,k_G,\alpha,\beta}$ and commits to the resulting string using the Merkle commitment scheme. Then τ vertices of the graph are randomly chosen. For each challenged vertex v, the prover computes and outputs the opening for this vertex as well as opening for all its predecessors. The verifier gets as input the identity, a commitment, and $\tau(d+1)$ openings, where d is the degree of the graph. It firstly verifies the consistency of all the openings with respect to the commitment. Secondly, it checks the local correctness of the $\mathcal{H}_{id}$-labeling.

The completeness of our scheme relies on the correctness of the underlying commitment scheme. The extractability will follow from the pebbling complexity of the graph $G_{\mu,k_G,\alpha,\beta}$ and the binding property of the commitment scheme. In particular, we prove the following statement:

Theorem 2. *Let $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{n_\mathcal{H}}$ be a random oracle, $G_{\mu,k_G,\alpha,\beta}$ be a SoLEG with $N = \mu(k_G+1)$ nodes and d in-degree, and $(\mathsf{MGen}^\mathcal{H}, \mathsf{MCommit}^\mathcal{H}, \mathsf{MOpen}^\mathcal{H}, \mathsf{MVer}^\mathcal{H})$ be a $(n_\mathcal{H}, n_\mathsf{cm}, N, n_\mathsf{op}, \nu_\mathsf{mt})$-Merkle commitment. Let $s, t \in \mathbb{N}$ be such that, for some $\delta \in [0, \beta - 2\alpha)$, we have $t < 2^{k_G}\alpha\mu$ and $s \le \delta\mu n_\mathcal{H}$. Then, the NIPoS scheme described in Fig. 4 is a $(s_\mathsf{P}, s_\mathsf{V}, s, t, k, n, \nu_\mathsf{pos})$-NIPoS for any $k \in \mathbb{N}$, as long as:*

$$\begin{aligned} s_\mathsf{P} &\ge k + n_\mathcal{H}(\mu + \log(N) + 1) + n \\ s \ge s_\mathsf{V} &\ge k + n + n_\mathcal{H} \\ n &= n_\mathsf{cm} + n_\mathsf{op}(d+1)(n_\mathcal{H}/\log(N)) \\ \nu_\mathsf{pos} &\le \exp\left(\frac{-n_\mathcal{H}\mu(\gamma - \delta)}{N\log(N)}\right) + \frac{n_\mathcal{H}(d+1)\nu_\mathsf{mt}}{\log(N)} + \frac{|\mathcal{Q}_\mathcal{H}(\mathsf{A})|}{2^{n_\mathcal{H}}}, \end{aligned}$$

where $\mathcal{Q}_\mathcal{H}(\mathsf{A})$ are the random oracle queries asked by A and $\gamma = \beta - 2\alpha$.

5.3 Security Analysis

In this section we show that the construction suggested by [40] satisfies the stronger form of soundness which we call extractability. In addition, we formalize the security analysis given in [40] and provide concrete parameters. Let us first explain the intuition behind the proof of Theorem 2. The adversary wins the game only if all the checked vertices have a correct $\mathcal{H}_{\tilde{id}}$-label. By the binding property of the underlying Merkle commitment scheme this means that the adversary A has to compute a partial $\mathcal{H}_{\tilde{id}}$-labeling of the graph $G_{\mu,k_G,\alpha,\beta}$. Since $\tilde{id}$ is not extractable from the query table of $\mathcal{Q}_\mathcal{H}(\mathsf{B})$ of the adversary B and it is not equal to id, the adversary A does not get any $\mathcal{H}_{\tilde{id}}$ label "for free" and hence, it has to compute the labeling on

[16] As recommended in [40] we will typically work with $0.7 \le \gamma \le 0.9$ to get loosely $40 < d < 200$.

NIPoS Construction

$\mathsf{Setup}^{\mathcal{H}}(1^\lambda)$**:** On input the security parameter λ, generate the public parameters $\omega_{\mathsf{cm}} \leftarrow \mathsf{MGen}^{\mathcal{H}}(1^\lambda)$ for the Merkle commitment. Consider the graph $G_{\mu,k_G,\alpha,\beta}$; recall that the number of nodes of $G_{\mu,k_G,\alpha,\beta}$ is given by $N = \mu(k_G+1)$ and the in-degree is $d \in O(1)$. Output $\omega_{\mathsf{pos}} = (G_{\mu,k_G,\alpha,\beta}, \omega_{\mathsf{cm}})$.

$\mathsf{P}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id)$**:** On input an identity $id \in \{0,1\}^k$, and the public parameters $\omega_{\mathsf{pos}} = (G_{\mu,k_G,\alpha,\beta}, \omega_{\mathsf{cm}})$, proceed as follows.

1. Generate a $\mathcal{H}_{id}$-labeling of $G_{\mu,k_G,\alpha,\beta}$. Denote the labeling by $\mathbf{z} = (z_1, \ldots, z_N)$, where each $z_i \in \{0,1\}^{n_{\mathcal{H}}}$.
2. Generate a commitment of $\mathbf{z}$, i.e. $\psi \leftarrow \mathsf{MCommit}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z})$ where $\psi \in \{0,1\}^{n_{\mathsf{cm}}}$.
3. Compute $\rho := \mathcal{H}(id, \psi)$. Using ρ as the randomness, pick τ vertices $\mathbf{v} = (v_1, v_2, \ldots, v_\tau)$ by setting $\mathbf{v} := \rho$ for $\tau = n_{\mathcal{H}}/\log(N)$, where each $v_i \in [N]$.
4. For each vertex $v_i \in \mathbf{v}$:
   (a) Compute the opening $(z_{v_i}, \phi_i) := \mathsf{MOpen}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}, v_i)$, for $(z_{v_i}, \phi_i) \in \{0,1\}^{n_{\mathsf{op}}}$.
   (b) Let $\Gamma^-(v_i) = (u_1^{(1)}, \ldots, u_d^{(i)})$ where each $u_j^{(i)} \in [N]$. Compute the opening corresponding to each $u_j^{(i)} \in \Gamma^-(v_i)$, i.e. $(z_{u_j^{(i)}}, \phi_j^{(j)}) := \mathsf{MOpen}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}, u_j^{(i)})$.
   (c) Define
   $$\pi_i := \Big((z_{v_i}, \phi_i), (z_{u_1^{(i)}}, \phi_1^{(i)}), \cdots, (z_{u_d^{(i)}}, \phi_d^{(i)})\Big) \in \{0,1\}^{n_{\mathsf{op}}(d+1)}.$$
5. Output $\pi := (\psi, (\pi_1, \ldots, \pi_\tau)) \in \{0,1\}^n$ as a proof of space for id.

$\mathsf{V}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id, \pi)$**:** On input the public parameters $\omega_{\mathsf{pos}} = (G_{\mu,k_G,\alpha,\beta}, \omega_{\mathsf{cm}})$, an identity $id \in \{0,1\}^k$, and a candidate proof of space $\pi \in \{0,1\}^n$, it first parses π as $(\psi, (\pi_1, \cdots, \pi_\tau))$, and computes $\rho := \mathcal{H}(id, \psi)$. Using ρ as the randomness, pick τ vertices $\mathbf{v} = (v_1, v_2, \ldots, v_\tau)$ by setting $\mathbf{v} := \rho$ for $\tau = n_{\mathcal{H}}/\log(N)$, where each $v_i \in [N]$ (exactly as the prover did). Hence, it proceeds as follows for each $i \in [\tau]$:

1. Parse $\pi_i := ((w_i, \phi_i), (w_1^{(i)}, \phi_1^{(i)}), \ldots, (w_d^{(i)}, \phi_d^{(i)}))$ and then:
   (a) Check that $w_i = \mathcal{H}(id, v_i, w_1^{(i)}, \ldots w_d^{(i)})$.
   (b) Check that $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(v_i, \psi, (w_i, \phi_i)) = 1$.
   (c) Let $\Gamma^-(v_i) := (u_1^{(i)}, \ldots, u_d^{(i)})$; for each $j \in [d]$ check that $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(u_j^{(i)}, \psi, (w_j^{(i)}, \phi_j^{(i)})) = 1$.
2. If the above checks succeed for all $i \in [\tau]$, then output 1, else output 0.

Figure 4: Our NIPoS construction.

its own. By Lemma 3, however, the labeling of the graph $G_{\mu,k_G,\alpha,\beta}$ requires either a lot of space or a lot of time neither of which the (s,t)-bounded adversary A has. Instead of computing all the labels correctly via random oracle calls, the adversary A can assign labels of some vertices to an arbitrary value which does not need to be computed and stored. However, if such partial labeling consists of too many faults, the probability that at least one of the faulty vertices will be checked is high. Consequently, a winning adversary can not save a lot of recourses by computing only a partial labeling of the graph.

Proof of Theorem 2. First notice that, by our construction, it is straightforward to see the bounds on $s_{\mathsf{P}}, s_{\mathsf{V}}$ and n. Here, we use Lemma 4 that there exists a poly-time algorithm that can label the graph $G_{\mu,k_G,\alpha,\beta}$ in $\mu n_{\mathcal{H}}$-bit space. Furthermore, since $s_{\mathsf{V}} \leq s < s_{\mathsf{P}}$ we get $k+n+n_{\mathcal{H}} \leq s$. Completeness is straightforward given these bounds and correctness of the Merkle commitment scheme. In the rest of the proof we will show the upper-bound on the soundness error ν_{pos}.

We start by constructing the knowledge extractor, taking as input the public parameters $\omega_{\mathsf{pos}} \in \{0,1\}^*$ and the sequence $\mathcal{Q}_{\mathcal{H}}(\mathsf{B})$ of B's random oracle queries and corresponding answers.

$\mathsf{K}(\omega_{\mathsf{pos}}, \mathcal{Q}_{\mathcal{H}}(\mathsf{B}))$:

- Let $\mathcal{IP}_{\mathcal{Q}_{\mathcal{H}}} = (u_1, \dots, u_{\mathsf{q_B}})$ be the input field of the query table.
- For each u_i, such that $|u_i| \geq k$, define $id(u_i) := (u_i[1], \dots, u_i[k])$; let $\mathcal{I}$ be the set that contains all unique k-bit strings $id(u_i)$.
- Output the set $\mathcal{I} = \{id_i\}_{i \in [q]}$.

Adapting to our construction we can describe the security game $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda)$ for any security parameter $\lambda \in \mathbb{N}$ as follows:

Game $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda)$:

1. Sample $\omega_{\mathsf{pos}} \leftarrow \mathsf{Setup}^{\mathcal{H}}(1^\lambda)$ and $\pi \leftarrow \mathsf{P}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id)$.
2. Let $\mathsf{A} \leftarrow \mathsf{B}^{\mathcal{H}}(\omega_{\mathsf{pos}}, id, \pi)$ and $\{id_i\}_{i \in [q]} := \mathsf{K}(\omega_{\mathsf{pos}}, \mathcal{Q}_{\mathcal{H}}(\mathsf{B}))$.
3. Let $(\tilde{id}, \tilde{\pi}) := \mathsf{A}^{\mathcal{H}}(id, \pi)$.
4. Parse $(\psi, (\pi_1, \cdots, \pi_\tau)) := \tilde{\pi}$ and compute $(v_1, \dots, v_\tau) := \mathcal{H}(\tilde{id}, \psi)$.
5. Output 1 if and only if, for every $i \in [\tau]$, the following holds. Assuming that $((w_i, \phi_i), (w_1^{(i)}, \phi_1^{(i)}), \dots, (w_d^{(i)}, \phi_d^{(i)})) := \pi_i$ and $(u_1^{(i)}, \dots, u_d^{(i)}) := \Gamma^-(v_i)$ all three conditions below are satisfied:
   (a) $w_i = \mathcal{H}(\tilde{id}, v_i, w_1^{(i)}, \dots w_d^{(i)})$;
   (b) $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(v_i, \psi, (w_i, \phi_i)) = 1$;
   (c) For each $j \in [d]$: $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(u_j^{(i)}, \psi, (w_j^{(i)}, \phi_j^{(i)})) = 1$.

Let us now fix an $id \in \{0,1\}^k$ and define the event UNBIND in the game $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda)$ adapted to our NIPoS construction (Fig. 4).

Event UNBIND: There exists a $\mathbf{z} := (z_1, \dots, z_N) \in \{0,1\}^{n_{\mathcal{H}}N}$ for which at least one of the following conditions is true:

1. $\exists i \in [\tau]$ such that $w_i \neq z_{v_i}$, $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(v_i, \psi, (z_{v_i}, \varphi_i)) = 1$ and $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(v_i, \psi, (w_i, \phi_i)) = 1$ where $(z_{v_i}, \varphi_i) := \mathsf{MOpen}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}, v_i)$.
2. $\exists i \in [\tau], j \in [d]$ such that $w_j^{(i)} \neq z_{u_j^{(i)}}$, $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(u_j^{(i)}, \psi, (z_{u_j^{(i)}}, \varphi_j^{(i)})) = 1$ and $\mathsf{MVer}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(u_j^{(i)}, \psi, (w_j^{(i)}, \phi_j^{(i)})) = 1$ where $(z_{u_j^{(i)}}, \varphi_j^{(i)}) := \mathsf{MOpen}^{\mathcal{H}}_{\omega_{\mathsf{cm}}}(\mathbf{z}, u_j^{(i)})$.

Assume that UNBIND happens for some adversary B. Then at least one of the two of above conditions must be true. For simplicity, we assume that the first condition is true; the proof for the case when the second condition holds follows similarly. We construct an algorithm $\mathsf{C}^{\mathcal{H},i}$ for each $i \in [\tau]$ which, hard-coded with polynomial-time algorithm A, wins the game $\mathbf{G}^{\mathsf{bind}}_{\mathsf{C},\mathbf{z},j}(\lambda)$ of the Merkle commitment scheme, where $j = v_i$, as follows:

Adversary $\mathsf{C}^{\mathcal{H},i}_{\mathsf{B},\mathbf{z}}$:

- Run $\omega_{\mathsf{pos}} \leftarrow \mathsf{Setup}^{\mathcal{H}}(1^{\lambda})$ and $\pi \leftarrow \mathsf{P}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(id)$ for the fixed id.
- Run $\mathsf{B}^{\mathcal{H}}(\omega_{\mathsf{pos}}, id, \pi)$. Answer RO queries made by B by querying the random oracle $\mathcal{H}$, and forwarding the answers to B.
- On receiving A from B, run $(\tilde{id}, \tilde{\pi}) := \mathsf{A}^{\mathcal{H}}(id, \pi)$ and parse $(\psi, (\pi_1, \cdots, \pi_\tau)) := \tilde{\pi}$ and $\pi_i := ((w_i, \phi_i), (w^{(i)}_1, \phi^{(i)}_1), \ldots, (w^{(i)}_d, \phi^{(i)}_d))$.
- Finally, return $(\psi, (w_i, \phi_i))$ as answer.

It is straightforward to observe that C perfectly simulates the view of B if the first condition is true for i. Therefore, using a simple union bound, we have that

$$\Pr[\textsc{Unbind}] \leq \sum_j \Pr[\mathbf{G}^{\mathsf{bind}}_{\mathsf{C},\mathbf{z},j}(\lambda) = 1] \leq \tau(d+1)\nu_{\mathsf{mt}}.$$

Hence,

$$\begin{aligned} \Pr[\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda) = 1] &\leq \Pr[\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda) = 1 \mid \neg\textsc{Unbind}] + \Pr[\textsc{Unbind}] \\ &\leq \underbrace{\Pr[\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda) = 1 \mid \neg\textsc{Unbind}]}_{:=\mathrm{E}} + \tau(d+1)\nu_{\mathsf{mt}} \end{aligned} \qquad (1)$$

Now, it is clear that the adversary can only win the game if it commits to a partial $\mathcal{H}_{\tilde{id}}$-labeling of the graph $G_{\mu,k_G,\alpha,\beta}$ where all checked vertices have the correct label. Hence, unless A successfully guesses the list of vertices ahead of time (which in turn requires predicting outputs of random oracle $\mathcal{H}$), it has to compute the partial labeling within its space-bound s. Let us now define the following event.

Event $\textsc{Guess}$: A guesses the output of at least one random oracle query correctly (i.e., without querying the random oracle).

If A makes $|\mathcal{Q}_{\mathcal{H}}(\mathsf{A})|$ queries to the random oracle in total, then clearly we have: $\Pr[\textsc{Guess}] \leq |\mathcal{Q}_{\mathcal{H}}(\mathsf{A})|/2^{n_{\mathcal{H}}}$. Hence:

$$\Pr[\mathrm{E}] \leq \Pr[\underbrace{\mathrm{E} \mid \neg\textsc{Guess}}_{:=\mathrm{E}'}] + \Pr[\textsc{Guess}] \leq \Pr[\mathrm{E} \mid \neg\textsc{Guess}] + |\mathcal{Q}_{\mathcal{H}}(\mathsf{A})|/2^{n_{\mathcal{H}}}. \qquad (2)$$

We have to bound the probability of E'. First, note that since we condition on both $\neg\textsc{Guess}$ and $\neg\textsc{Unbind}$ we have the following observations about E':

1. $\neg\textsc{Unbind}$ implies that if the adversary A successfully opens some commitment $\tilde{\psi}$ to some value z with respect to some index i (i.e., $\mathsf{MVer}^{\mathcal{H}}$ succeeds), then it must have committed to a string $\mathbf{z} = (z_1, \ldots, z_\tau)$ such that $z_i = z$.

2. However, instead of labeling every node of $G_{\mu,k_G,\alpha,\beta}$ via random oracle queries, A can also put some faults (some arbitrary value that requires no space/time) to some node and "hope" that it is not challenged on those nodes. Since we assume $\neg\textsc{Guess}$, A can only win whenever no v_i in $(v_1, \ldots, v_\tau)$ is labeled with faults, because v_i's are decided randomly where the randomness comes from the output of $\mathcal{H}$.

We prove the following claim.

Claim 1. *Suppose* A *wins by computing a* $\mathcal{H}_{\tilde{id}}$*-labeling of* $G_{\mu,k_G,\alpha,\beta}$ *with* m *faults when conditioned on* $\neg$Guess *and* $\neg$Unbind*. Then we conclude:*

- A *does not have any initial* $\mathcal{H}_{\tilde{id}}(\cdot)$*-labels of the graph* $G_{\mu,k_G,\alpha,\beta}$*.*
- $\Pr[\mathrm{E}'] \leq \exp(\frac{-\tau \cdot m}{N})$.

Proof. To prove the first item observe that if the adversary A wins the game $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},id}(\lambda)$, then $\tilde{id}$ is not extractable by K from the random oracle queries made by the adversary B in the "pre-computation phase", and $\tilde{id} \neq id$. In particular, this means that A does not have any initial $\mathcal{H}_{\tilde{id}}(\cdot)$-label as, without guessing, any such labeling can only be done via a valid query to $\mathcal{H}_{\tilde{id}}(\cdot)$ (note that we assume $\neg$Guess).

To prove the second item we observe that A can only win whenever no v_i in $(v_1, \ldots, v_\tau)$ is labeled with faults. Let us call that event NoFault. We have that:

$$\Pr[\mathrm{E}'] \leq \Pr[\textsc{NoFault}] \leq \left(1 - \frac{m}{N}\right)^\tau \leq \exp\left(-\frac{\tau \cdot m}{N}\right),$$

which proves the claim. □

Next, let us consider two cases based on the bound on m. Define $\zeta := \gamma - \delta$. Then we have:

Case 1: $m \geq \zeta\mu$. In this case by the above claim we directly get that $\Pr[\mathrm{E}'] \leq \exp\left(\frac{-n_{\mathcal{H}}\mu(\gamma-\delta)}{N\log(N)}\right)$ that proves the theorem.

Case 2: $m < \zeta\mu$. In this case we first apply Lemma 1 to get that if A (that is (s,t)-bounded) computes a $\mathcal{H}_{\tilde{id}}(\cdot)$-labeling with m faults, then there exists an $(s + n_{\mathcal{H}}m, t)$-bounded polynomial-time adversary A' that computes the $\mathcal{H}_{\tilde{id}}$ labeling of the graph $G_{\mu,k_G,\alpha,\beta}$ with no faults and m initial labels.

Further, applying Lemma 2 we obtain that there exists a pebbling strategy $\mathbf{P} = (P_0, \ldots, P_{t(\mathbf{P})})$ with $|P_0| = m < \gamma\mu$ initial pebbles, total space complexity $s(\mathbf{P}) \leq \frac{(s+n_{\mathcal{H}}m)}{n_{\mathcal{H}}} \leq \gamma\mu$ and time complexity $t(\mathbf{P}) \leq t$ that pebbles the entire graph $G_{\mu,k_G,\alpha,\beta}$. In particular, since $1 - \zeta \geq \alpha$,[17] it pebbles at least $\mu - m$ $(= \mu(1-\zeta) \geq \alpha\mu)$ sinks which were not initially pebbled.[18] Finally, applying Lemma 3 we can have $t \geq t(\mathbf{P}) \geq 2^{k_G}\alpha\mu$ that is a contradiction.

□

Using the parameters from Theorem 2 we obtain the following corollary.

Corollary 1. *Let* $\lambda \in \mathbb{N}$ *be a security parameter. Let* $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{n_{\mathcal{H}}}$ *be a random oracle,* $G_{\mu,k_G,\alpha,\beta}$ *be a SoLEG with* $N = \mu(k_G + 1)$ *nodes and* $d = O(1)$ *in-degree, and* $(\mathsf{MGen}^{\mathcal{H}}, \mathsf{MCommit}^{\mathcal{H}}, \mathsf{MOpen}^{\mathcal{H}}, \mathsf{MVer}^{\mathcal{H}})$ *be a* $(n_{\mathcal{H}}, n_{\mathsf{cm}}, N, n_{\mathsf{op}}, \nu_{\mathsf{mt}})$*-Merkle commitment such that:*

$$n_{\mathcal{H}} = \lambda^2 \qquad \gamma = \beta - 2\alpha \in (0,1) \qquad k_G = \lambda - 1 \qquad \mu = \lambda^3$$
$$n_{\mathsf{cm}} = \lambda^2 \qquad n_{\mathsf{op}} = O(\lambda^2\log(\lambda)) \qquad \nu_{\mathsf{mt}} \in \mathrm{negl}(\lambda).$$

Then, for any $\delta \in (0,\gamma)$*, the scheme described in Fig. 4 is a* $(s_{\mathsf{P}}, s_{\mathsf{V}}, s, t, k, n, \nu_{\mathsf{pos}})$*-NIPoS, for* $t \in \mathrm{poly}(\lambda)$ *and*

$$k = O(\lambda^4) \qquad s_{\mathsf{P}} = O(\lambda^5) \qquad s_{\mathsf{V}} = O(\lambda^4)$$
$$n = O(\lambda^4) \qquad O(\lambda^4) \leq s \leq \delta \cdot \lambda^5 \qquad \nu_{\mathsf{pos}} \leq \exp\left(\frac{-(\gamma-\delta)\lambda}{\log(\lambda)}\right) + \mathrm{negl}(\lambda) \in \mathrm{negl}(\lambda)$$

[17] This can be observed by noticing that $1 - \zeta \geq \alpha \Leftrightarrow 1 \geq \beta - \alpha - \delta$ which is always true as $\alpha, \beta, \delta \in (0,1)$.

[18] Since we assume that A' starts with m initial labels, it is possible that all the labels are located the last layer of $G_{\mu,k_G,\alpha,\beta}$, that is the sinks.

6 Our Coding Scheme

In this section we construct a leaky space-bounded non-malleable code based on any NIPoS with appropriate parameters. Our construction and its security are stated in Section 6.1; the proof appears in Section 6.2. In Section 6.3, we instantiate our construction by plugging the concrete parameters from NIPoS construction described in Section 5.

6.1 Code Construction

Let $(\mathsf{Setup}^{\mathcal{H}}, \mathsf{P}^{\mathcal{H}}, \mathsf{V}^{\mathcal{H}})$ be a NIPoS in the ROM where $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{n_{\mathcal{H}}}$ denotes the random oracle for some $n_{\mathcal{H}} \in \text{poly}(\lambda)$. We define a (k, n)-coding scheme $\Pi = (\mathsf{Init}^{\mathcal{H}}, \mathsf{Encode}^{\mathcal{H}}, \mathsf{Decode}^{\mathcal{H}})$ as follows.

$\mathsf{Init}^{\mathcal{H}}(1^\lambda)$**:** Given as input a security parameter λ, it generates the public parameters for the NIPoS as $\omega_{\mathsf{pos}} \leftarrow \mathsf{Setup}^{\mathcal{H}}(1^\lambda)$, and outputs $\omega := \omega_{\mathsf{pos}}$.

$\mathsf{Encode}^{\mathcal{H}}_{\omega}(x)$**:** Given as input the public parameters $\omega = \omega_{\mathsf{pos}}$ and a message $x \in \{0,1\}^k$, it runs the prover to generate the proof of space $\pi \leftarrow \mathsf{P}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(x)$ using the message x as identity. Then it outputs $c := (x, \pi) \in \{0,1\}^n$ as a codeword.

$\mathsf{Decode}^{\mathcal{H}}_{\omega}(c)$**:** Given a codeword c, it first parses c as (x, π). Then it runs the verifier $b := \mathsf{V}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(x, \pi)$. If $b = 1$ it outputs x, otherwise it outputs $\perp$.

Theorem 3. *Let λ be a security parameter. Suppose that $(\mathsf{Setup}^{\mathcal{H}}, \mathsf{P}^{\mathcal{H}}, \mathsf{V}^{\mathcal{H}})$ is a $(s_{\mathsf{P}}, s_{\mathsf{V}}, s, k_{\mathsf{pos}}, n_{\mathsf{pos}}, \text{negl}(\lambda))$-NIPoS. Then, for any $p \in \mathbb{N}$ such that $k_{\mathsf{pos}} + n_{\mathsf{pos}} \leq p \leq s$ and $\theta \in \text{poly}(\lambda)$, the (k, n)-code $\Pi = (\mathsf{Init}^{\mathcal{H}}, \mathsf{Encode}^{\mathcal{H}}, \mathsf{Decode}^{\mathcal{H}})$ defined above is an $(\ell, s, p, \theta, s_{\mathsf{V}})$-SP-NMC in the ROM, where*

$$k = k_{\mathsf{pos}} \qquad n = k_{\mathsf{pos}} + n_{\mathsf{pos}} \qquad \ell = \theta \cdot O(\log \lambda).$$

Recall that, in our definition of non-malleability, the parameter s represents the space available for tampering, which is split into two components: p bits of persistent space, which includes the n bits necessary for storing the codeword and which is never erased, and $s-p$ bits of transient space that is erased after each tampering query.

Also, note that the above statement shows a clear tradeoff between the parameter θ (controlling the number of allowed tampering queries) and the leakage bound ℓ. Indeed, the larger θ, the more leakage we need, until the security guarantee becomes empty; this tradeoff is consistent with Theorem 1 (see also Fig. 1), as we know that leaky space-bounded non-malleability, for non-trivial values of ℓ, is impossible for $p \approx n + k$, whenever $\theta \geq k$.

6.2 Proof of Security

The correctness of the coding scheme is guaranteed by the perfect completeness of the NIPoS. Moreover, since the decoding algorithm simply runs the verifier of the NIPoS, it is straightforward to observe that decoding is s_{V} bounded.

Auxiliary algorithms. We start by introducing two auxiliary algorithms that will be useful in the proof. Recall that, by extractability of the NIPoS, there exists a deterministic polynomial-time algorithm K such that, given the public parameters ω_{pos} and a table of RO queries $\mathcal{Q}_{\mathcal{H}}$, returns a set of identities $\{id_i\}_{i \in [q]}$, for some $q \in \text{poly}(\lambda)$. We define the following algorithms that use K as a subroutine.

Algorithm $\mathsf{Find}(\omega_{\mathsf{pos}}, id, \mathcal{Q}_{\mathcal{H}})$**:** Given a value $id \in \{0,1\}^{k_{\mathsf{pos}}}$, it first runs K to obtain $\{id_i\}_{i\in[q]} := \mathsf{K}(\omega_{\mathsf{pos}}, \mathcal{Q}_{\mathcal{H}})$. If there exists an index $i \in [q]$ such that $id = id_i$, then it returns the string $\mathsf{str} := \mathsf{bit}(i)||01$,[19] where the function $\mathsf{bit}(\cdot)$ returns the binary representation of its input. Otherwise, the algorithm returns the flag 1^ℓ. Clearly, $\ell = \lceil\log(q)\rceil + 2$.

Algorithm $\mathsf{Reconstruct}(\omega_{\mathsf{pos}}, \mathsf{str}, \mathcal{Q}_{\mathcal{H}})$**:** On receiving an ℓ-bit string str and a RO query table $\mathcal{Q}_{\mathcal{H}}$, it works as follows depending on the value of str:

- If $\mathsf{str} = 0^\ell$, output the symbol $\mathsf{same}^\star$.
- If $\mathsf{str} = 1^\ell$, output the symbol $\perp$.
- If $\mathsf{str} = a||01$, set $i := \mathsf{bit}^{-1}(a)$. Hence, run algorithm K to get the set $\{id_i\}_{i\in[q]} := \mathsf{K}(\omega_{\mathsf{pos}}, \mathcal{Q}_{\mathcal{H}})$; in case $i \in [q]$, output the value $x := id_i$, otherwise output $\perp$.
- Else, output $\perp$.

Constructing the simulator. We now describe the simulator $\mathsf{S}^{\mathsf{D}} = (\mathsf{S}_1^{\mathsf{D}}, \mathsf{S}_2^{\mathsf{D}})$, depending on a PPT distinguisher D.[20] A formal description of the simulator is given in Fig. 5; we provide some intuitions below.

Informally, algorithm S_1 simulates the random oracle $\mathcal{H}$ by sampling a random key $\chi \leftarrow \{0,1\}^{n_{\mathsf{key}}}$ for a pseudorandom function (PRF) $\mathsf{PRF}_\chi : \{0,1\}^* \to \{0,1\}^{n_{\mathcal{H}}}$; hence, it defines $\mathcal{H}(u) := \mathsf{PRF}_\chi(u)$ for any $u \in \{0,1\}^*$.[21] S_2 receives the description of the RO (i.e., the PRF key χ) from S_1, and for each tampering query A_i from D it asks a leakage query L_i to its leakage oracle. The leakage query hard-codes the description of the simulated RO, the table $\mathcal{Q}_{\mathcal{H}}(\mathsf{D})$ consisting of all RO queries asked by D (until this point), and the code of all tampering algorithms $\mathsf{A}_1, \cdots, \mathsf{A}_i$. Thus, L_i first encodes the target message x to generate a codeword c, applies the composed function $\mathsf{A}_i \circ \mathsf{A}_{i-1} \circ \cdots \circ \mathsf{A}_1$ on c to generate the tampered codeword $\tilde{c}_i$, and decodes $\tilde{c}_i$ obtaining a value $\tilde{x}_i$. Finally, the leakage function signals whether $\tilde{x}_i$ is equal to the original message x, to $\perp$, or to some of the identities the extractor K would output given the list of D's RO queries (as defined in algorithm Find). Upon receiving the output from the leakage oracle, S_2 runs $\mathsf{Reconstruct}$ and outputs whatever this algorithm returns.

Some intuitions. Firstly, note that in the real experiment the random oracle is a truly random function, whereas in the simulation random oracle queries are answered using a PRF. However, using the security of the PRF, we can move to a mental experiment that is exactly the same as the simulated game, but replaces the PRF with a truly random function.

Secondly, a closer look at the algorithms Find and $\mathsf{Reconstruct}$ reveals that the only case in which the simulation strategy goes wrong is when the tampered codeword $\tilde{c}_i$ is valid, but the leakage corresponding to the output of Find provokes a $\perp$ by $\mathsf{Reconstruct}$ for some $i \in [\theta]$. We denote this event as FAIL. We prove that FAIL occurs exactly when the adversary D violates the extractibility property of the underlying NIPoS, which happens only with negligible probability.

To simplify the notation in the proof, let us write

$$\mathsf{D}^{\mathrm{cnm}} := \mathsf{D}^{\mathcal{H}(\cdot),\mathcal{O}_{\mathrm{cnm}}^{\Pi,x,\omega,s,p}(\cdot)},\ \mathsf{D}^{\mathrm{sim}'} := \mathsf{D}^{\mathcal{H}(\cdot),\mathcal{O}_{\mathrm{sim}}^{\mathsf{S}_2,\ell,x,s,\omega}(\cdot)},\ \mathsf{D}^{\mathrm{sim}} := \mathsf{D}^{\mathsf{S}_1(\cdot),\mathcal{O}_{\mathrm{sim}}^{\mathsf{S}_2,\ell,x,s,\omega}(\cdot)}$$

to denote the interaction in the real, resp. mental, resp. simulated experiment.

[19] Looking ahead, in the simulation we use the strings 0^ℓ and 1^ℓ as flags; therefore, appending 01 to str ensures that str is never misinterpreted as those flags.

[20] In the rest of the proof we drop the superscript D, and just let $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$.

[21] Such a PRF can be instantiated using any PRF with fixed domain, and then applying the standard Merkle-Damgård transformation to extend the input domain to arbitrary-length strings.

Simulator $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$

1. Let $\mathsf{PRF}_\chi : \{0,1\}^* \to \{0,1\}^{n_\mathcal{H}}$ be a PRF. The simulator S_1 samples a uniform random key $\chi \leftarrow \{0,1\}^{n_\mathsf{key}}$ and defines $\mathcal{H} := \mathsf{PRF}_\chi$. The query table $\mathcal{Q}_\mathcal{H}(\mathsf{D})$ that stores RO queries from D is initially empty.
2. For $i \in [\theta]$ the simulator does the following:
   (a) S_1 simulates the random oracle queries made by D, before A_i is chosen, and updates the table $\mathcal{Q}_\mathcal{H}(\mathsf{D})$ accordingly.
   (b) On receiving the adversary A_i, the simulator S_2 queries its leakage oracle $\mathcal{O}^{\ell,x}_{\mathrm{leak}}$ with $L : \{0,1\}^k \to \{0,1\}^\ell$ (where $\ell = O(\log(\lambda))$) described as follows:

      Description of L:
      - L is hard-coded with the description of $\mathcal{H}$ (i.e., with PRF_χ), the table $\mathcal{Q}_\mathcal{H}(\mathsf{D})$, the code of $(\mathsf{A}_1, \mathsf{A}_2, \dots, \mathsf{A}_i)$, and the code of the knowledge extractor K of the NIPoS.
      - Produce the codeword $c \leftarrow \mathsf{Encode}^\mathcal{H}_\omega(x)$ and initialize the auxiliary space $\sigma := 0^{s-n}$.
      - Let $\widetilde{\mathsf{A}}_i := \mathsf{A}_i \circ \mathsf{A}_{i-1} \circ \cdots \circ \mathsf{A}_1$. Run $\widetilde{\mathsf{A}}_i$ to get $(\tilde{c}, \tilde{\sigma}) := \widetilde{\mathsf{A}}_i(c; \sigma)$.
      - Compute $\tilde{x} := \mathsf{Decode}^\mathcal{H}_\omega(\tilde{c})$. If $\tilde{x} = \perp$, output the flag 1^ℓ, else, if $\tilde{x} = x$, output the flag 0^ℓ; otherwise run $\mathsf{str}_{\tilde{x}} := \mathsf{Find}(\omega, \tilde{x}, \mathcal{Q}_\mathcal{H}(\mathsf{D}))$ and output $\mathsf{str}_{\tilde{x}}$.
      - All other oracle queries that are not asked by D (e.g., queries made by A_j, or while running Encode etc.) are simulated internally.

   (c) On receiving an ℓ-bit string str from L, simulator S_2 runs $\tilde{x} \leftarrow \mathsf{Reconstruct}(\omega, \mathsf{str}_{\tilde{x}}, \mathcal{Q}_\mathcal{H}(\mathsf{D}))$ and outputs $\tilde{x}$.

Figure 5: Description of the simulator $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$

Formal analysis. Consider an adversary D which makes θ queries to $\mathcal{O}_{\mathrm{cnm}}$. By Definition 7, we need to prove that the simulator $\mathsf{S}^\mathsf{D} = (\mathsf{S}_1^\mathsf{D}, \mathsf{S}_2^\mathsf{D})$ defined in Fig. 5 is such that, for all values $x \in \{0,1\}^k$, there is a negligible function $\nu : \mathbb{N} \to [0,1]$ satisfying

$$\left|\Pr\left[\mathsf{D}^{\mathrm{cnm}}(\omega) = 1 :\ \omega \leftarrow \mathsf{Init}^\mathcal{H}(1^\lambda)\right] - \Pr\left[\mathsf{D}^{\mathrm{sim}}(\omega) = 1 :\ \omega \leftarrow \mathsf{Init}^{\mathsf{S}_1}(1^\lambda)\right]\right| \leq \nu(\lambda).$$

A straightforward reduction to the pseudorandomness of the PRF yields:

$$\left|\Pr\left[\mathsf{D}^{\mathrm{sim}}(\omega) = 1 :\ \omega \leftarrow \mathsf{Init}^{\mathsf{S}_1}(1^\lambda)] - \Pr[\mathsf{D}^{\mathrm{sim}'}(\omega) = 1 :\ \omega \leftarrow \mathsf{Init}^\mathcal{H}(1^\lambda)\right]\right| \leq \nu'(\lambda),$$

where $\nu' : \mathbb{N} \to [0,1]$ is a negligible function.

Let us now fix some arbitrary $x \in \{0,1\}^k$. For every $i \in [\theta]$, we recursively define the event $\textsc{NotExtr}_i$ as:

$$\begin{aligned}\textsc{NotExtr}_i := \neg\textsc{NotExtr}_{i-1}\ \wedge\ &\mathsf{Decode}^\mathcal{H}_\omega(\tilde{c}) \notin \{\perp, x\}\\ &\wedge\ \mathsf{Find}(\omega, \mathsf{Decode}^\mathcal{H}_\omega(\tilde{c}), \mathcal{Q}_\mathcal{H}(\mathsf{D})) = 1^\ell,\end{aligned}$$

where $\textsc{NotExtr}_0$ is an empty event that never happens and $(\tilde{c}, \tilde{\sigma}) := \widetilde{\mathsf{A}}_i(c, \sigma)$ for $\widetilde{\mathsf{A}}_i := \mathsf{A}_i \circ \mathsf{A}_{i-1} \circ \cdots \circ \mathsf{A}_1$. In other words, the event $\textsc{NotExtr}_i$ happens when A_i is the first adversary that tampers to a valid codeword of a message $\tilde{x} \neq x$ which is not extraxtable from $\mathcal{Q}_{\mathcal{H}}(\mathsf{D})$. In addition, we define the event

$$\textsc{Fail} := \bigvee_{i \in [\theta]} \textsc{NotExtr}_i.$$

Now, we can bound the probability that D succeeds as follows:

$$\begin{aligned}
&\left|\Pr\left[\mathsf{D}^{\mathrm{cnm}}(\omega) = 1\right] - \Pr\left[\mathsf{D}^{\mathrm{sim}'}(\omega) = 1\right]\right| \qquad (3)\\
\leq\ &\left|\Pr\left[\mathsf{D}^{\mathrm{cnm}}(\omega) = 1 \mid \neg\textsc{Fail}\right] - \Pr\left[\mathsf{D}^{\mathrm{sim}'}(\omega) = 1 \mid \neg\textsc{Fail}\right]\right| \cdot \Pr[\neg\textsc{Fail}]\\
&+ \left|\Pr\left[\mathsf{D}^{\mathrm{cnm}}(\omega) = 1 \mid \textsc{Fail}\right] - \Pr\left[\mathsf{D}^{\mathrm{sim}'}(\omega) = 1 \mid \textsc{Fail}\right]\right| \cdot \Pr[\textsc{Fail}]\\
\leq\ &\left|\Pr\left[\mathsf{D}^{\mathrm{cnm}}(\omega) = 1 \mid \neg\textsc{Fail}\right] - \Pr\left[\mathsf{D}^{\mathrm{sim}'}(\omega) = 1 \mid \neg\textsc{Fail}\right]\right| + \Pr[\textsc{Fail}],
\end{aligned}$$

where in the above equations the probability is taken also on the sampling of $\omega \leftarrow \mathsf{Init}^{\mathcal{H}}(1^\lambda)$. We complete the proof by showing the following two claims.

Claim 2. *Event* Fail *happens with negligible probability.*

Proof. Assume that for some $x \in \{0,1\}^k$ adversary D provokes the event Fail with non-negligible probability. This implies that there is at least one index $j \in [\theta]$ such that event $\textsc{NotExtr}_j$ happens with non-negligible probability. We construct an efficient algorithm B running in game $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},x}(\lambda)$, that attempts to break the extractability of the NIPoS:

Algorithm $\mathsf{B}^{\mathcal{H}}_{\mathsf{D}}$:

1. Receive as input $\omega_{\mathsf{pos}} \leftarrow \mathsf{Setup}^{\mathcal{H}}(1^\lambda)$, $x \in \{0,1\}^{k_{\mathsf{pos}}}$, and $\pi \leftarrow \mathsf{P}^{\mathcal{H}}_{\omega_{\mathsf{pos}}}(x)$.
2. Assign $(c, \sigma) := (x || \pi, 0^{s-n})$, $\mathcal{Q}_{\mathcal{H}}(\mathsf{D}) := \emptyset$, and define $\mathsf{A} := \mathsf{Id}$, where $\mathsf{Id} \colon \{0,1\}^s \to \{0,1\}^s$ is the identity function.
3. For $i \in [\theta]$ proceed as follows:
   (a) Answer random oracle queries made by D, before A_i is chosen, by querying $\mathcal{H}$ in game $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},x}(\lambda)$ and forwarding the answers to D; in addition, store these queries in the table $\mathcal{Q}_{\mathcal{H}}(\mathsf{D})$.
   (b) On receiving A_i, set $\mathsf{A} := \mathsf{A} \circ \mathsf{A}_i$ and run $(\tilde{c}, \tilde{\sigma}) := \mathsf{A}_i(c; \sigma)$.
   (c) Compute $\tilde{x} := \mathsf{Decode}^{\mathcal{H}}_{\omega}(\tilde{c})$ and run $\mathsf{str}_{\tilde{x}} := \mathsf{Find}(\omega, \tilde{x}, \mathcal{Q}_{\mathcal{H}}(\mathsf{D}))$. If $\mathsf{str}_{\tilde{x}} = 1^\ell$ and $\tilde{x} \neq \perp$, then output A and stop. Otherwise send $\tilde{x}$ to D and let $(c, \sigma) := (\tilde{c}, \tilde{\sigma}_0 || 0^{s-p})$, where $\tilde{\sigma}_0 || \tilde{\sigma}_1 := \tilde{\sigma}$.

We observe that B perfectly simulates the view of $\mathsf{D}^{\mathrm{sim}'}$. So, if there exists at least one $j \in [\theta]$ for which $\textsc{NotExtr}_j$ happens, B wins the game $\mathbf{G}^{\mathsf{ext}}_{\mathsf{B},x}(\lambda)$. Therefore we have that:

$$\Pr[\mathbf{G}^{\mathsf{ext}}_{\mathsf{D},x}(\lambda) = 1] \geq \Pr[\exists j \in [\theta] :\ \textsc{NotExtr}_j]$$

which, combined with the extractability of NIPoS, completes the proof. □

Claim 3. $\left|\Pr\left[\mathsf{D}^{\mathrm{cnm}}(1^\lambda) = 1 \mid \neg\textsc{Fail}\right] - \Pr\left[\mathsf{D}^{\mathrm{sim}'}(1^\lambda) = 1 \mid \neg\textsc{Fail}\right]\right| = 0$

Proof. By inspection of the simulator's description it follows that, conditioning on event Fail not happening, the simulation oracle using S_2 yields a view that is identical to the one obtained when interacting with the tampering oracle. The claim follows. □

Combining the above two claims together with Eq. (3), we obtain that

$$\left|\Pr\left[\mathsf{D}^{\mathrm{cnm}}(\omega) = 1 :\ \omega \leftarrow \mathsf{Init}^{\mathcal{H}}(1^\lambda)\right] - \Pr\left[\mathsf{D}^{\mathrm{sim}'}(\omega) = 1 :\ \omega \leftarrow \mathsf{Init}^{\mathsf{S}_1}(1^\lambda)\right]\right|$$

is negligible, as desired.

It remains to argue about the size of leakage. To this end, it suffices to note that the simulator S_2 receives $O(\log(\lambda))$ bits of leakage for every $i \in [\theta]$. Thus, the total amount of leakage is $\theta \cdot O(\log(\lambda))$, exactly as stated in the theorem.

6.3 Concrete Instantiation and Parameters

Instantiating Theorem 3 with our concrete NIPoS from Corollary 1, and using bounds from Theorem 1, we obtain the following corollaries. The first corollary provides an upper bound on the number of tolerated tampering queries at the price of a high (but still non-trivial) leakage parameter.

Corollary 2. *For any $\gamma, \delta, \varepsilon \in (0, 1)$, there exists an explicit construction of a (k, n)-code in the ROM that is a $(\gamma \cdot k, s, p, \theta, \Theta(\lambda^4))$-SP-NMC, where*

$$k = \Theta(\lambda^4) \qquad n = \Theta(\lambda^4) \qquad \Theta(\lambda^4) \leq p \leq s = \delta\lambda^5 \quad \theta = \Theta(\lambda^{4-\varepsilon}).$$

The second corollary yields a smaller number of tolerated tampering queries with optimal (logarithmic) leakage parameter.

Corollary 3. *For any $\delta \in (0, 1)$, there exists an explicit construction of a (k, n)-code in the ROM that is an $(O(\log\lambda), s, p, \theta, O(\lambda^4))$-SP-NMC, where*

$$k = O(\lambda^4) \qquad n = O(\lambda^4) \qquad O(\lambda^4) \leq p \leq s = \delta\lambda^5 \quad \theta = O(1).$$

7 Trading Leakage for Tamper-Proof Security

We revise the standard application of non-malleable codes to obtain protection against memory tampering attacks. The main idea, put forward in [21], is very simple. Let $\mathcal{F}$ be an arbitrary functionality, initialized with "secret key" κ; instead of storing κ, we store an encoding c of κ, computed via a non-malleable code. Hence, whenever we have to run $\mathcal{F}$, we decode c obtaining a value $\tilde{\kappa}$ which we use to evaluate the functionality on any chosen input. It is not too hard to show that this idea yields security against tampering attacks against the secret key, for the same class of adversaries supported by the non-malleable code.

This methodology, also known as "tamper simulatability", has been explored in several variants [35, 25, 13, 26]. Here, we propose yet another variant where the above compiler is instantiated using a leaky space-bounded continuously non-malleable code; this yields security in a model where it is possible to "trade" security against space-bounded memory tampering, with some bits of leakage on the secret key, an idea already explored in a related line of research [28].

7.1 Leaky Tamper Simulatability

Let $\mathcal{F} : \{0,1\}^k \times \{0,1\}^{k_{\mathsf{in}}} \to \{0,1\}^{k_{\mathsf{out}}}$ be a randomized functionality, taking as input a secret value $\kappa \in \{0,1\}^k$ and a string $m \in \{0,1\}^{k_{\mathsf{in}}}$, and producing a value $y \leftarrow \mathcal{F}(\kappa, m) \in \{0,1\}^{k_{\mathsf{out}}}$. For simplicity, we consider the case of stateless functionalities where the value κ is never updated during the computation; an extension to the case of stateful functionalities is immediate, along the lines of previous work [21, 35, 25]. We note, however, that since updating the value κ

requires execution of the encoding algorithm (which uses a lot of space), considering only stateless functionalities is natural in our model.

Given a non-malleable code Π, the hardened functionality corresponding to $\mathcal{F}$ is defined below. For consistency with the rest of the paper, we state the definition in the ROM.

Definition 10 (Hardened functionality)**.** Consider a functionality $\mathcal{F} : \{0,1\}^k \times \{0,1\}^{k_{\mathsf{in}}} \to \{0,1\}^{k_{\mathsf{out}}}$, and let $\Pi = (\mathsf{Init}^{\mathcal{H}}, \mathsf{Encode}^{\mathcal{H}}, \mathsf{Decode}^{\mathcal{H}})$ be a (k,n)-code in the ROM. For parameters $s,p \in \mathbb{N}$, with $s \geq p \geq n$, the (s,p)-memory hardened functionality $\hat{\mathcal{F}}(\Pi,s,p)$ corresponding to $\mathcal{F}$ consists of algorithms $(\mathsf{Setup}^{\mathcal{H}}, \mathsf{Run}^{\mathcal{H}})$ with the following syntax.

- $\mathsf{Setup}^{\mathcal{H}}(1^\lambda, s, \kappa)$: Upon input the security parameter $\lambda \in \mathbb{N}$, sample $\omega \leftarrow \mathsf{Init}^{\mathcal{H}}(1^\lambda)$, let $c \leftarrow \mathsf{Encode}^{\mathcal{H}}_\omega(\kappa)$, and set $\mathcal{M} := c||0^{p-n}||0^{s-p}$. Output $(\omega, \mathcal{M})$.
- $\mathsf{Run}^{\mathcal{H}}_\omega(\mathcal{M}, m)$: Parse $\mathcal{M} := c||\sigma_0||\sigma_1$ and let $\tilde{\kappa} = \mathsf{Decode}^{\mathcal{H}}_\omega(c)$. If $\tilde{\kappa} = \perp$, set $\tilde{y} = \perp$; else, run $\tilde{y} \leftarrow \mathcal{F}(\tilde{\kappa}, m)$. Update $\mathcal{M} := c||\sigma_0||0^{s-p}$ and output $(\tilde{y}, \mathcal{M})$.

It follows by correctness of the encoding scheme that, for all inputs, $\hat{\mathcal{F}}(\Pi,s,p)$ computes exactly the same functionality as $\mathcal{F}$. Notice that the hardened functionality corresponding to $\mathcal{F}$ has p bits of persistent storage (i.e., n bits for storing the secret encoding and $p-n$ bits for auxiliary data); the remaining $s-p$ bits represent transient storage that is needed for decoding the codeword and running the original functionality with the obtained key (this memory is erased after each evaluation).

In case there is not enough transient space to decode or to run the original functionality, an external memory must be used. Thus, we get a natural trade-off between the amount of auxiliary data that can be stored on the device and the class of functionalities that can be executed without using an external memory.

Tampering experiment. To define security, we consider an s-bounded adversary that tampers with the memory content of the hardened functionality. This is done via the experiment described below, which is executed by a PPT algorithm D, and is parametrized by an (s,p)-memory hardened functionality $\hat{\mathcal{F}}(\Pi,s,p)$, a key $\kappa \in \{0,1\}^k$, a parameter $\theta \in \mathbb{N}$, and security parameter $\lambda \in \mathbb{N}$.

<u>Experiment **TamperInteract**$(\mathsf{D}, \hat{\mathcal{F}}(\Pi,s,p), \kappa, \theta, \lambda)$:</u>

1. Run $(\omega, \mathcal{M}) \leftarrow \mathsf{Setup}^{\mathcal{H}}(1^\lambda, s, \kappa)$ and give ω to D.
2. D can run the following commands (in an arbitrary order):
   - $\langle \mathtt{Tamper}, \mathsf{A} \in \mathcal{A}^s_{\text{space}} \rangle$: Parse $\mathcal{M} := c||\sigma_0||\sigma_1$. Let $(\tilde{c}, \tilde{\sigma}_0, \tilde{\sigma}_1) = \mathsf{A}(c; \sigma_0||\sigma_1)$, and update $\mathcal{M} := \tilde{c}||\tilde{\sigma}_0||\tilde{\sigma}_1$. This command can be run for at most θ times.
   - $\langle \mathtt{Execute}, m \in \{0,1\}^{k_{\mathsf{in}}} \rangle$: Execute $(\tilde{y}, \mathcal{M}) \leftarrow \mathsf{Run}^{\mathcal{H}}_\omega(\mathcal{M}, m)$, and return $\tilde{y}$. This command can be executed an arbitrary polynomial number of times.
   - $\langle \mathtt{RO}, u \in \{0,1\}^* \rangle$: Return $v = \mathcal{H}(u)$. This command can be executed an arbitrary polynomial number of times.
3. D outputs a bit as a function of its view.

Leaky simulation. Intuitively, a non-malleable code is ℓ-leaky tamper simulatable if the above tampering experiment can be simulated with black-box access to the original functionality $\mathcal{F}$, plus ℓ bits of leakage on the secret key. This is formalized in the experiment described below, which is executed by a PPT algorithm D and is parametrized by a functionality $\mathcal{F}$, a PPT simulator S, a value $\ell \in \mathbb{N}$, an initial key $\kappa \in \{0,1\}^k$, a parameter $\theta \in \mathbb{N}$, and security parameter $\lambda \in \mathbb{N}$.

Experiment $\mathbf{BBLeak}(\mathsf{D}, \mathcal{F}, \mathsf{S}, \ell, \kappa, \theta, \lambda)$:

1. The simulator S, which is given black-box access to $\mathcal{F}(\kappa, \cdot)$ and oracle access to $\mathcal{O}_{\text{leak}}^{\ell,\kappa}(\cdot)$, emulates the entire view of D. In particular:
   - It takes care of simulating the public parameters and answering (polynomially many) random oracle queries;
   - It needs to answer (at most θ) tampering queries and (polynomially many) execute queries.
2. D outputs a bit as a function of its view.[22]

Definition 11 (Leaky tamper simulatability)**.** Let $\ell, s, p, \theta, k, n \in \mathbb{N}$ be functions of the security parameter $\lambda \in \mathbb{N}$, with $s \geq p \geq n$. A (k, n)-code Π is ℓ-leaky (s, p)-space θ-tamper simulatable in the ROM, if for all PPT distinguishers D there exists a PPT simulator S such that for all functionalities $\mathcal{F}$, and for all $\kappa \in \{0,1\}^k$, there is a negligible function $\nu : \mathbb{N} \to [0,1]$ for which

$$\Big| \Pr\Big[\mathbf{TamperInteract}(\mathsf{D}, \hat{\mathcal{F}}(\Pi, s, p), \kappa, \theta, \lambda) = 1\Big] \\ - \Pr\left[\mathbf{BBLeak}(\mathsf{D}, \mathcal{F}, \mathsf{S}, \ell, \kappa, \theta, \lambda) = 1\right] \Big| \leq \nu(\lambda).$$

7.2 Analysis

In the following theorem we show the correspondence between leaky non-malleable and tamper simulatable codes.

Theorem 4. *Let Π be an ℓ-leaky (s, p)-space-bounded θ-continuously non-malleable code in the ROM. Then, Π is also ℓ-leaky (s, p)-space θ-tamper simulatable in the ROM.*

Informally, Theorem 4 states that every functionality $\mathcal{F}$ that is resistant to ℓ bits of leakage on the secret key can be protected against memory tampering by an ℓ-leaky non-malleable code.

Proof. We start by describing the simulator $\hat{\mathsf{S}}$, which is based on the simulator $\mathsf{S} := (\mathsf{S}_1, \mathsf{S}_2)$ of the underlying non-malleable code. Without loss of generality, we assume that each command of the form $\langle \texttt{Tamper}, \cdot \rangle$ is followed by at least one or more commands of the form $\langle \texttt{Execute}, \cdot \rangle$.[23]

Simulator $\hat{\mathsf{S}}(1^\lambda, \mathsf{S}_1, \mathsf{S}_2)$:

- Run $\omega \leftarrow \mathsf{Init}^{\mathsf{S}_1}(1^\lambda)$ and forward ω to D.
- Upon input a command of the form $\langle \texttt{Tamper}, \mathsf{A} \in \mathcal{A}_{\text{space}}^s \rangle$, invoke $\mathsf{S}_2(1^\lambda, \omega, \mathsf{A})$; whenever S_2 asks for a leakage query, send the same query to the oracle $\mathcal{O}_{\text{leak}}^{\ell,\kappa}(\cdot)$ and forward the answer back to S_2. Let $\tilde{\kappa} \in \{0,1\}^k \cup \{\mathsf{same}^\star, \bot\}$ be the value returned by the simulator.
- Upon input a command of the form $\langle \texttt{Execute}, m \rangle$, proceed as follows:
  - If $\tilde{\kappa} = \bot$, return $\tilde{y} := \bot$;
  - If $\tilde{\kappa} = \mathsf{same}^\star$, forward m to the black-box functionality $\mathcal{F}(\kappa, \cdot)$, receive back the answer y, and output such value;
  - Else, if $\tilde{\kappa} \in \{0,1\}^k$, return $\tilde{y} \leftarrow \mathcal{F}(\tilde{\kappa}, m)$.

[22] Typically, the simulator is restricted to run the black-box functionality on the very same inputs on which the distinguisher specifies its execute queries.

[23] If this is not the case, S can just combine the different space-bounded algorithms into one algorithm.

- Upon input a command of the form $\langle \texttt{RO}, u \rangle$, return the same as $\mathsf{S}_1(u)$.

Assume there exists a PPT distinguisher $\hat{\mathsf{D}}$, a value $\kappa \in \{0,1\}^k$, and a polynomial $\rho(\lambda)$, such that, for infinitely many values of $\lambda \in \mathbb{N}$, we have

$$\Big|\Pr\Big[\mathbf{TamperInteract}(\hat{\mathsf{D}}, \hat{\mathcal{F}}(\Pi, s, p), \kappa, \theta, \lambda) = 1\Big] \\ - \Pr\Big[\mathbf{BBLeak}(\hat{\mathsf{D}}, \mathcal{F}, \hat{\mathsf{S}}, \ell, \kappa, \theta, \lambda) = 1\Big]\Big| \geq 1/\rho(\lambda).$$

We construct a PPT distinguisher D, asking at most θ tampering queries, such that

$$\Big|\Pr\Big[\mathsf{D}^{\mathcal{H}(\cdot), \mathcal{O}_{\mathrm{cnm}}^{\Pi,\kappa,\omega,s,p}(\cdot)}(\omega) = 1 : \ \omega \leftarrow \mathsf{Init}^{\mathcal{H}}(1^\lambda)\Big] \\ - \Pr\Big[\mathsf{D}^{\mathsf{S}_1(\cdot), \mathcal{O}_{\mathrm{sim}}^{\mathsf{S}_2,\ell,\kappa,s,\omega}(\cdot)}(\omega) = 1 : \ \omega \leftarrow \mathsf{Init}^{\mathsf{S}_1}(1^\lambda)\Big]\Big| \geq 1/\rho(\lambda),$$

which contradicts ℓ-leaky (s,p)-space-bounded θ-continuous non-malleability of the underlying coding scheme. A description of D (running $\hat{\mathsf{D}}$) follows below.

Distinguisher D:

- Receive the public parameters $\omega \in \{0,1\}^*$, and initialize $\hat{\mathsf{D}}(\omega)$.
- Upon input a command of the form $\langle \texttt{Tamper}, \mathsf{A} \in \mathcal{A}^s_{\mathrm{space}} \rangle$, forward (a description of) A to the target tampering oracle (either $\mathcal{O}_{\mathrm{cnm}}$ or $\mathcal{O}_{\mathrm{sim}}$) and receive back a value $\tilde{\kappa} \in \{0,1\}^k \cup \{\bot\}$.
- Upon input a command of the form $\langle \texttt{Execute}, m \rangle$, if $\tilde{\kappa} \neq \bot$ answer with $y \leftarrow \mathcal{F}(\tilde{\kappa}, m)$, else answer with $\bot$.
- Upon input a command of the form $\langle \texttt{RO}, u \rangle$, forward u to the target random oracle (either $\mathcal{H}$ or S_1) and forward the answer $v \in \{0,1\}^{n_{\mathcal{H}}}$ to $\hat{\mathsf{D}}$.
- Output the same as $\hat{\mathsf{D}}$ outputs.

For the analysis, first note that D is almost as efficient as $\hat{\mathsf{D}}$ and moreover, depending on its target random oracle being either equal to $\mathcal{H}$ or S_1, the simulation of random oracle queries performed by D is identical to either the one in experiment **TamperInteract** or the one in experiment **BBLeak**. As for the simulation of tampering/execute queries, we observe the following:

- If D's target tampering oracle is $\mathcal{O}_{\mathrm{cnm}}$, each space-bounded algorithm A is applied to the current state $\texttt{st} = (c, \sigma_0, \sigma_1)$—playing the role of the memory $\mathcal{M}$ of the hardened functionality—yielding a modified state $(\tilde{c}, \tilde{\sigma}_0, \tilde{\sigma}_1)$; hence, the value $\tilde{\kappa} = \mathsf{Decode}_\omega(\tilde{c})$ is received by D, and the memory is updated to $(\tilde{c}, \tilde{\sigma}_0, 0^{s-p})$ (i.e., the transient memory is erased).

  When an execute query is run later on, say upon input a message $m \in \{0,1\}^{k_{\mathsf{in}}}$, the distinguisher runs $\mathcal{F}$ upon input $\tilde{\kappa}$ and m (unless $\tilde{\kappa} = \bot$, in which case $\bot$ is returned). By inspection, this is exactly what happens in experiment **TamperInteract**, and thus the view of $\hat{\mathsf{D}}$ is perfectly simulated.

- If D's target tampering oracle is $\mathcal{O}_{\mathrm{sim}}$, each space-bounded algorithm A causes an invocation of simulator S_2, yielding a value $\tilde{\kappa} \in \{0,1\}^k \cup \{\bot\}$ that is received by D. (Recall that, in case S_2 outputs $\mathsf{same}^\star$, the simulation oracle replaces this value with the hard-wired value κ.)

When an execute query is run later on, say upon input a message $m \in \{0,1\}^{k_{\mathsf{in}}}$, the distinguisher runs $\mathcal{F}$ upon input $\tilde{\kappa}$ and m (unless $\tilde{\kappa} = \bot$, in which case $\bot$ is returned). By inspection, this is exactly what happens in experiment **BBLeak**, and thus the view of $\hat{\mathsf{D}}$ is perfectly simulated.

We conclude that D makes a perfect simulation, and thus retains the same (non-negligible) distinguishing advantage as that of $\hat{\mathsf{D}}$. This finishes the proof. □

References

[1] Divesh Aggarwal, Yevgeniy Dodis, Tomasz Kazana, and Maciej Obremski. Non-malleable reductions and applications. In *STOC*, pages 459–468, 2015.

[2] Divesh Aggarwal, Yevgeniy Dodis, and Shachar Lovett. Non-malleable codes from additive combinatorics. In *STOC*, pages 774–783, 2014.

[3] Ross Anderson and Markus Kuhn. Tamper resistance: a cautionary note. In *WOEC*, Berkeley, CA, USA, 1996. USENIX Association.

[4] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. In *SCN*, pages 538–557, 2014.

[5] Marshall Ball, Dana Dachman-Soled, Mukul Kulkarni, and Tal Malkin. Non-malleable codes for bounded depth, bounded fan-in circuits. In *EUROCRYPT*, pages 881–908, 2016.

[6] Rishiraj Bhattacharyya and Pratyay Mukherjee. Non-adaptive programmability of random oracle. *Theor. Comput. Sci.*, 592:97–114, 2015.

[7] Dan Boneh, Henry Corrigan-Gibbs, and Stuart E. Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *ASIACRYPT*, pages 220–248, 2016.

[8] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2):101–119, 2001.

[9] Eshan Chattopadhyay, Vipul Goyal, and Xin Li. Non-malleable extractors and codes, with their many tampered extensions. In *ACM STOC*, pages 285–298, 2016.

[10] Mahdi Cheraghchi and Venkatesan Guruswami. Capacity of non-malleable codes. In *Innovations in Theoretical Computer Science*, pages 155–168, 2014.

[11] Sandro Coretti, Yevgeniy Dodis, Björn Tackmann, and Daniele Venturi. Non-malleable encryption: Simpler, shorter, stronger. In *TCC*, pages 306–335, 2016.

[12] Sandro Coretti, Ueli Maurer, Björn Tackmann, and Daniele Venturi. From single-bit to multi-bit public-key encryption via non-malleable codes. In *TCC*, pages 532–560, 2015.

[13] Dana Dachman-Soled, Feng-Hao Liu, Elaine Shi, and Hong-Sheng Zhou. Locally decodable and updatable non-malleable codes and their applications. In *TCC*, pages 427–450, 2015.

[14] Ivan Damgård, Sebastian Faust, Pratyay Mukherjee, and Daniele Venturi. The chaining lemma and its application. In *Information Theoretic Security*, pages 181–196, 2015.

[15] Ivan Damgård, Sebastian Faust, Pratyay Mukherjee, and Daniele Venturi. Bounded tamper resilience: How to go beyond the algebraic barrier. *J. Cryptology*, 30(1):152–190, 2017.

[16] Yevgeniy Dodis and Yu Yu. Overcoming weak expectations. In *TCC*, pages 1–22, 2013.

[17] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In *CRYPTO*, pages 37–54, 2005.

[18] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *CRYPTO*, pages 585–605, 2015.

[19] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-evolution schemes resilient to space-bounded leakage. In *CRYPTO*, pages 335–353, 2011.

[20] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. One-time computable self-erasing functions. In *TCC*, pages 125–143, 2011.

[21] Stefan Dziembowski, Krzysztof Pietrzak, and Daniel Wichs. Non-malleable codes. In *Innovations in Computer Science*, pages 434–452, 2010.

[22] Antonio Faonio, Jesper Buus Nielsen, and Daniele Venturi. Mind your coins: Fully leakage-resilient signatures with graceful degradation. In *ICALP*, pages 456–468, 2015.

[23] Antonio Faonio, Jesper Buus Nielsen, and Daniele Venturi. Fully leakage-resilient signatures revisited: Graceful degradation, noisy leakage, and construction in the bounded-retrieval model. *Theor. Comput. Sci.*, 660:23–56, 2017.

[24] Antonio Faonio and Daniele Venturi. Efficient public-key cryptography with bounded leakage and tamper resilience. In *ASIACRYPT*, pages 877–907, 2016.

[25] Sebastian Faust, Pratyay Mukherjee, Jesper Buus Nielsen, and Daniele Venturi. Continuous non-malleable codes. In *TCC*, pages 465–488, 2014.

[26] Sebastian Faust, Pratyay Mukherjee, Jesper Buus Nielsen, and Daniele Venturi. A tamper and leakage resilient von Neumann architecture. In *PKC*, pages 579–603, 2015.

[27] Sebastian Faust, Pratyay Mukherjee, Daniele Venturi, and Daniel Wichs. Efficient non-malleable codes and key derivation for poly-size tampering circuits. *IEEE Trans. Information Theory*, 62(12):7179–7194, 2016.

[28] Sebastian Faust, Krzysztof Pietrzak, and Daniele Venturi. Tamper-proof circuits: How to trade leakage for tamper-resilience. In *ICALP*, pages 391–402, 2011.

[29] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[30] Marc Fischlin, Anja Lehmann, Thomas Ristenpart, Thomas Shrimpton, Martijn Stam, and Stefano Tessaro. Random oracles with(out) programmability. In *ASIACRYPT*, pages 303–320, 2010.

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

[32] Carmit Hazay, Adriana López-Alt, Hoeteck Wee, and Daniel Wichs. Leakage-resilient cryptography from minimal assumptions. In *EUROCRYPT*, pages 160–176, 2013.

[33] Zahra Jafargholi and Daniel Wichs. Tamper detection and continuous non-malleable codes. In *TCC*, pages 451–480, 2015.

[34] Jonathan Katz and Vinod Vaikuntanathan. Signature schemes with bounded leakage resilience. In *ASIACRYPT*, pages 703–720, 2009.

[35] Feng-Hao Liu and Anna Lysyanskaya. Tamper and leakage resilience in the split-state model. In *CRYPTO*, pages 517–532, 2012.

[36] Ralph C. Merkle. Method of providing digital signatures. US Patent 4309569, January 5 1982.

[37] Pratyay Mukherjee. *Protecting Cryptographic Memory against Tampering Attack*. PhD thesis, Aarhus University, 2015.

[38] Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. *SIAM J. Comput.*, 41(4):772–814, 2012.

[39] Jesper Buus Nielsen, Daniele Venturi, and Angela Zottarel. Leakage-resilient signatures with graceful degradation. In *PKC*, pages 362–379, 2014.

[40] Ling Ren and Srinivas Devadas. Proof of space from stacked expanders. In *TCC*, pages 262–285, 2016.