Locally Decodable and Updatable Non-Malleable Codes and Their Applications

Dana Dachman-Soled
University of Maryland
danadach@ece.umd.edu

Feng-Hao Liu
University of Maryland
fenghao@cs.umd.edu

Elaine Shi
University of Maryland
elaine@cs.umd.edu

Hong-Sheng Zhou
Virginia Commonwealth University
hszhou@vcu.edu

August 25, 2014

Abstract

Non-malleable codes, introduced as a relaxation of error-correcting codes by Dziembowski, Pietrzak and Wichs (ICS '10), provide the security guarantee that the message contained in a tampered codeword is either the same as the original message or is set to an unrelated value. Various applications of non-malleable codes have been discovered, and one of the most significant applications among these is the connection with tamper-resilient cryptography. There is a large body of work considering security against various classes of tampering functions, as well as non-malleable codes with enhanced features such as *leakage resilience*.

In this work, we propose combining the concepts of *non-malleability*, *leakage resilience*, and *locality* in a coding scheme. The contribution of this work is three-fold:

1. As a conceptual contribution, we define a new notion of *locally decodable and updatable non-malleable code* that combines the above properties.
2. We present two simple and efficient constructions achieving our new notion with different levels of security.
3. We present an important application of our new tool – securing RAM computation against memory tampering and leakage attacks. This is analogous to the usage of traditional non-malleable codes to secure implementations in the *circuit* model against memory tampering and leakage attacks.

Contents

1 Introduction

The notion of non-malleable codes was defined by Dziembowski, Pietrzak and Wichs [23] as a relaxation of error-correcting codes. Informally, a coding scheme is **non-malleable** against a tampering function if by tampering with the codeword, the function can either keep the underlying message unchanged or change it to an unrelated message. Designing non-malleable codes is not only an interesting mathematical task, but also has important implications in cryptography; for example, Coretti et al. [13] showed an efficient construction of a mulit-bit CCA secure encryption scheme from a single-bit one via non-malleable codes. Agrawal et al. [3] showed how to use non-malleable codes to build non-malleable commitments. Most notably, the notion has a deep connection with security against so-called *physical attacks*; indeed, using non-malleable codes to achieve security against physical attacks was the original motivation of the work [23]. Due to this important application, research on non-malleable codes has become an important agenda, and drawn much attention in both coding theory and cryptography.

Briefly speaking, physical attacks target implementations of cryptographic algorithms beyond their input/output behavior. For example, researchers have demonstrated that leaking/tampering with sensitive secrets such as cryptographic keys, through timing channel, differential power analysis, and various other attacks, can be devastating [41, 42, 6, 7, 2, 33, 47], and therefore the community has focused on developing new mechanisms to defend against such strong adversaries [36, 29, 45, 35, 22, 46, 28, 17, 31, 37, 27, 19, 20, 14, 48, 32, 16, 40, 15, 18]. Dziembowski, Pietrzak and Wichs [23] showed a simple and elegant mechanism to secure implementations against memory tampering attacks by using non-malleable codes – instead of storing the secret (in the clear) on a device, one instead stores an encoding of the secret. The security of the non-malleable code guarantees that the adversary cannot learn more than what can be learnt via black box access to the device, even though the adversary may tamper with memory.

In a subsequent work, Liu and Lysyanskaya [44] extended the notion to capture **leakage resilience** as well – in addition to non-malleability, the adversary cannot learn anything about the underlying message even while obtaining partial leakage of the codeword. By using the approach outlined above, one can achieve security guarantees against both tampering and leakage attacks. In recent years, researchers have been studying various flavors of non-malleable codes; for example some work has focused on constructions against different classes of tampering functions, some has focused on different additional features, (e.g. continual attacks, rates of the scheme, etc), and some focused on other applications [11, 21, 26, 24, 9, 10, 1, 3].

In this paper, we focus on another important feature inspired from the field of coding theory – **locality**. More concretely, we consider a coding scheme that is locally decodable and updatable. As introduced by Katz and Trevisan [38], local decodability means that in order to retrieve a portion of the underlying message, one does not need to read through the whole codeword. Instead, one can just read a few locations at the codeword. Similarly, local updatability means that in order to update some part of the underlying messages, one only needs to update some parts of the codeword. Locally decodable codes have many important applications in private information retrieval [12] and secure multi-party computation [34], and have deep connections with complexity theory; see [51]. Achieving local decodability and updatability simultaneously makes the task more challenging. Recently, Chandran et al. [8] constructed a locally decodable and updatable code in the setting of *error-correcting* codes. They also show an application to dynamic proofs of retrievability. Motivated by the above results, we further ask the following intriguing question:

Can we build a coding scheme enjoying all three properties, i.e., non-malleability, leakage resilience, and locality? If so, what are its implications in cryptography?

Our Results. In light of the above questions, our contribution is three-fold:

- **(Notions).** We propose new notions that combine the concepts of non-malleability, leakage resilience, and locality in codes. First, we formalize a new notion of *locally decodable and updatable non-malleable codes* (against one-time attacks). Then, we extend this new notion to capture leakage resilience under continual attacks.
- **(Constructions).** We present two simple constructions achieving our new notions. The first construction is highly efficient—in order to decode (update) one block of the encoded messages, only *two* blocks of the codeword must be read (written)—but is only secure against *one-time* attacks. The second construction achieves security against *continual* attacks, while requiring $\log(n)$ number of reads (writes) to perform one decode (update) operation, where n is the number of blocks of the underlying message.
- **(Application).** We present an important application of our new notion – achieving tamper and leakage resilience in the random access machine (RAM) model. We first define a new model that captures tampering and leakage attacks in the RAM model, and then give a generic compiler that uses our new notion as a main ingredient. The compiled machine will be resilient to leakage and tampering on the random access memory. This is analogous to the usage of traditional non-malleable codes to secure implementations in the *circuit* model.

1.1 Techniques

In this section, we present a technical overview of our results.

Locally Decodable Non-malleable Codes. Our first goal is to consider a combination of concepts of non-malleability and local decodability. Recall that a coding scheme is non-malleable with respect to a tampering function f if the decoding of the tampered codeword remains the same or becomes some unrelated message. To capture this idea, the definition in the work [23] requires that there exists a simulator (with respect to such f) who outputs same^* if the decoding of the tampered codeword remains the same as the original one, or he outputs a decoded message, which is unrelated to the original one. In the setting of local decodability, we consider encodings of blocks of messages $M = (m_1, m_2, \ldots, m_n)$, and we are able to retrieve m_i by running $\textsc{dec}^{\textsc{enc}(M)}(i)$, where the decoding algorithm gets oracle access to the codeword.

The combination faces a subtlety that we cannot directly use the previous definition: suppose a tampering function f only modifies one block of the codeword, then it is likely that $\textsc{dec}$ remains unchanged for most places. (Recall a $\textsc{dec}$ will only read a few blocks of the codeword, so it may not detect the modification.) In this case, the (overall) decoding of $f(C)$ (i.e. $(\textsc{dec}^{f(C)}(1), \ldots, \textsc{dec}^{f(C)}(n))$) can be highly related to the original message, which intuitively means it is highly malleable.

To handle this issue, we consider a more fine-grained experiment. Informally, we require that for any tampering function f (within some class), there exists a simulator that computes a vector of decoded messages $\vec{m}^*$, a set of indices $\mathcal{I} \subseteq [n]$. Here $\mathcal{I}$ denotes the coordinates of the underlying messages that have been tampered with. If $\mathcal{I} = [n]$, then the simulator thinks that the decoded messages are $\vec{m}^*$, which should be unrelated to the original messages. On the other hand, if $\mathcal{I} \subsetneq [n]$, the simulator thinks that all the messages not in $\mathcal{I}$ remain unchanged, while those in $\mathcal{I}$ become $\perp$. This intuitively means the tampering function can do only one of the following cases:

1. It destroys a block (or blocks) of the underlying messages while keeping the other blocks unchanged, or

2. If it modifies a block of the underlying messages to some unrelated string, then it must have modified all blocks of the underlying messages to encodings of unrelated messages.

Our construction of locally decodable non-malleable code is simple – we use the idea similar to the *key encapsulation mechanism/data encapsulation mechanism (KEM/DEM) framework*. Let NMC be a regular non-malleable code, and $\mathcal{E}$ be a secure (symmetric-key) authenticated encryption. Then to encode blocks of messages $M = (m_1, \ldots, m_n)$, we first sample a secret key sk of $\mathcal{E}$, and output $(\mathsf{NMC.ENC}(\mathsf{sk}), \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m_1, 1), \ldots, \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m_n, n))$. The intuition is clear: if the tampering function does not change the first block, then by security of the authenticated encryption, any modification of the rest will become $\perp$. (Note that here we include a tag of positions to prevent permutation attacks). On the other hand, if the tampering function modified the first block, it must be decoded to an unrelated secret key sk'. Then by semantic security of the encryption scheme, the decoded values of the rest must be unrelated. The code can be updated locally: in order to update m_i to some m_i', one just need to retrieve the 1^{st} and $(i+1)^{\text{st}}$ blocks. Then he just computes a *fresh* encoding of $\mathsf{NMC.ENC}(\mathsf{sk})$ and the ciphertext $\mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m_i')$, and writes back to the same positions.

Extensions to Leakage Resilience against Continual Attacks. We further consider a notion that captures leakage attacks in the continual model. First we observe that suppose the underlying non-malleable code is also leakage resilient [44], the above construction also achieves one-time leakage resilience. Using the same argument of Liu and Lysyanskaya [44], if we can refresh the whole encoding, we can show that the construction is secure against continual attacks. However, in our setting, refreshing the whole codeword is not counted as a solution since this is in the opposite of the spirit of our main theme – *locality*. The main challenge is how to refresh (update) the codeword locally while maintaining tamper and leakage resilience.

To capture the local refreshing and continual attacks, we consider a new model where there is an updater $\mathcal{U}$ who reads the whole underlying messages and decides how to update the codeword (using the local update algorithm). The updater is going to interact with the codeword in a continual manner, while the adversary can launch tampering and leakage attacks between two updates. To define security we require that the adversary cannot learn anything of the underlying messages via tampering and leakage attacks from the interaction.

We note that if there is no update procedure at all, then no coding scheme can be secure against continual leakage attacks if the adversary can learn the whole codeword bit-by-bit. In our model, the updater and the adversary take turns interacting with the codeword – the adversary tampers with and/or gets leakage of the codeword, and then the updater *locally* updates the codeword, and the process repeats. See Section 2 for the formal model.

Then we consider how to achieve this notion. First we observe that the construction above is not secure under continual attacks: suppose by leakage the adversary can get a full ciphertext $\mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m_i, i)$ at some point, and then the updater updates the underlying message to m_i'. In the next round, the adversary can apply a *rewind attack* that modifies the codeword back with the old ciphertext. Under such attack, the underlying messages have been modified to some related messages. Thus the construction is not secure.

One way to handle this type of rewind attacks is to tie all the blocks of ciphertexts together with a "time stamp" that prevents the adversary from replacing the codeword with old ciphertexts obtained from leakage. A straightforward way is to hash all the blocks of encryptions using a collision resistant hash function and also encode this value into the non-malleable code, i.e., $C = (\mathsf{NMC.ENC}(\mathsf{sk}, v), \mathcal{E}.\mathsf{Encrypt}(1, m_1), \ldots, \mathcal{E}.\mathsf{Encrypt}(n, m_n))$, where $v = h(\mathcal{E}.\mathsf{Encrypt}(1, m_1), \ldots, \mathcal{E}.\mathsf{Encrypt}(n, m_n))$. Intuitively, suppose the adversary replaces a block

$\mathcal{E}.\mathsf{Encrypt}(i, m_i)$ by some old ciphertexts, then it would be caught by the hash value v unless he tampered with the non-malleable code as well. But if he tampers with the non-malleable code, the decoding will be unrelated to sk, and thus the rest of ciphertexts become "un-decryptable". This approach prevents the rewind attacks, yet it does not preserve the local properties, i.e. to decode a block, one needs to check the consistency of the hash value v, which needs to read all the blocks of encryptions. To prevent the rewind attacks while maintaining local decodability/updatability, we use the Merkle tree technique, which allows local checks of consistency.

The final encoding outputs $(\mathsf{NMC}.\textsc{enc}(\mathsf{sk}, v), \mathcal{E}.\mathsf{Encrypt}(1, m_1), \ldots, \mathcal{E}.\mathsf{Encrypt}(n, m_n), T)$, where T is the Merkle tree of $(\mathcal{E}.\mathsf{Encrypt}(1, m_1), \ldots, \mathcal{E}.\mathsf{Encrypt}(n, m_n))$, and v is its root (it can also be viewed as a hash value). To decode a position i, the algorithm reads the 1^{st}, and the $(i+1)^{\text{st}}$ blocks together with a path in the tree. If the path is inconsistent with the root, then output $\perp$. To update, one only needs to re-encode the first block with a new root, and update the $(i+1)^{\text{st}}$ block and the tree. We note that Merkle tree allows local updates: if there is only one single change at a leaf, then one can compute the new root given only a path passing through the leaf and the root. So the update of the codeword can be done locally by reading the 1^{st}, the $(i+1)^{\text{st}}$ blocks and the path. We provide a detailed description and analysis in Section 3.3.

Application to Tamper and Leakage Resilient RAM Model of Computation. Whereas regular non-malleable codes yield secure implementations against memory tampering in the circuit model, our new tool yields secure implementations against memory tampering (and leakage) in the RAM model.

In our RAM model, the data and program to be executed are stored in the random access memory. Through a CPU with a small number of (non-persistent) registers[1], execution proceeds in clock cycles: In each clock-cycle memory addresses are read and stored in registers, a computation is performed, and the contents of the registers are written back to memory. In our attack model, we assume that the CPU circuitry (including the non-persistent registers) is secure – the computation itself is not subject to physical attacks. On the other hand, the random access memory, and the memory addresses are prone to leakage and tampering attacks. We remark that if the CPU has *secure* persistent registers that store a secret key, then the problem becomes straightforward: Security can be achieved using encryption and authentication together with oblivious RAM [30]. We emphasize that in our model, persistent states of the CPU are stored in the memory, which are prone to leakage and tampering attacks. As our model allows the adversary to learn the access patterns the CPU made to the memory, together with the leakage and tampering power on the memory, the adversary can somewhat learn the messages transmitted over the *bus* or tamper with them (depending on the attack classes allowed on the memory). For simplicity of presentation, we do not define attacks on the bus, but just remark that these attacks can be implicitly captured by learning the access patterns and attacking the memory[2].

In our formal modeling, we consider a next instruction function Π, a database D (stored in the random access memory) and an internal state (using the non-persistent registers). The CPU will interact (i.e., read/write) with the memory based on Π, while the adversary can launch tamper and leakage attacks during the interaction.

Our compiler is very simple, given the ORAM technique and our new codes as building

[1] These non-persistent registers are viewed as part of the circuitry that stores some transient states while the CPU is computing at each cycle. The number of these registers is small, and the CPU needs to erase the data in order to reuse them, so they cannot be used to store a secret key that is needed for a long term of computation.

[2] There are some technical subtleties to simulate all leakage/tampering attacks on the values passing the bus using memory attacks (and addresses). We defer the rigorous treatment to future work.

blocks. Informally speaking, given any next instruction function Π and database D, we first use ORAM technique to transform them into a next instruction function $\widetilde{\Pi}$ and a database $\widetilde{D}$. Next, we use our local non-malleable code $(\textsc{enc}, \textsc{dec}, \textsc{update})$ to encode $\widetilde{D}$ into $\widehat{D}$; the compiled next instruction function $\widehat{\Pi}$ does the following: run $\widetilde{\Pi}$ to compute the next "virtual" read/write instruction, and then run the local decoding or update algorithms to perform the physical memory access.

Intuitively, the inner ORAM protects leakage of the address patterns, and the outer local non-malleable codes prevent an attacker from modifying the contents of memory to some *different* but *related* value. Since at each cycle the CPU can only read and write at a small number of locations of the memory, using regular non-malleable codes does not work. Our new notion of locally decodable and updatable non-malleable codes exactly solves these issues!

1.2 Related Work

Different flavors of non-malleable codes were studied [23, 11, 44, 21, 26, 24, 9, 10, 1, 3]. We can use these constructions to secure implementations against memory attacks in the circuit model, and also as our building block for the locally decodable/updatable non-malleable codes. See also Section 3.4 for further exposition.

Securing circuits or CPUs against physical attacks is an important task, but out of the scope of this paper. Some partial results can be found in previous work [43, 49, 36, 45, 35, 22, 46, 28, 17, 31, 37, 27, 19, 20, 14, 48, 32, 50, 40, 15, 18].

In an independent and concurrent work, Faust et al. [25] also considered securing RAM computation against tampering and leakage attacks. We note that both their model and techniques differ considerably from ours. In the following, we highlight some of these differences. The main focus of [25] is constructing RAM compilers for keyed functions, denoted $\mathcal{G}_{\mathsf{K}}$, to allow secure RAM emulation of these functions in the presence of leakage and tampering. In contrast, our work focuses on construcing compilers that transform any dynamic RAM machine into a RAM machine secure against leakage and tampering. Due to this different perspective, our compiler explicitly utilizes an underlying ORAM compiler, while they assume that the memory access pattern of input function $\mathcal{G}$ is independent of the secret state K (e.g., think of $\mathcal{G}$ as the circuit representation of the function). In addition to the split-state tampering and leakage attacks considered by both papers, [25] do not assume that memory can be overwritten or erased, but require the storage of a tamper-proof program counter. With regard to techniques, they use a stronger version of non-malleable codes in the split-state setting (called continual non-malleable codes [24]) for their construction. Finally, in their construction, each memory location is encoded using an expensive non-malleable encoding scheme, while in our construction, non-malleable codes are used only for a small portion of the memory, while highly efficient symmetric key authenticated encryption is used for the remainder.

2 Locally Decodable and Updatable Non-Malleable Codes

In this section, we first review the concepts of non-malleable (leakage resilient) codes. Then we present our new notion that combines non-malleability, leakage resilience, and locality.

2.1 Preliminary

Definition 2.1 (Coding Scheme)**.** *Let $\Sigma, \hat{\Sigma}$ be sets of strings, and $\kappa, \hat{\kappa} \in \mathbb{N}$ be some parameters. A coding scheme consists of two algorithms* $(\textsc{enc}, \textsc{dec})$ *with the following syntax:*

– *The encoding algorithm (*perhaps randomized*) takes input a block of message in* Σ *and outputs a codeword in* $\hat{\Sigma}$.
– *The decoding algorithm takes input a codeword in* $\hat{\Sigma}$ *and outputs a block of message in* Σ.

We require that for any message $m \in \Sigma$, $\Pr[\textsc{dec}(\textsc{enc}(m)) = m] = 1$, *where the probability is taken over the choice of the encoding algorithm. In binary settings, we often set* $\Sigma = \{0,1\}^{\kappa}$ *and* $\hat{\Sigma} = \{0,1\}^{\hat{\kappa}}$.

Definition 2.2 (Non-malleability [23])**.** *Let* k *be the security parameter,* $\mathcal{F}$ *be some family of functions. For each function* $f \in \mathcal{F}$, *and* $m \in \Sigma$, *define the tampering experiment:*

$$\mathbf{Tamper}_m^f \stackrel{\text{def}}{=} \left\{ \begin{array}{c} c \leftarrow \textsc{enc}(m), \tilde{c} := f(c), \tilde{m} := \textsc{dec}(\tilde{c}). \\ Output: \ \tilde{m}. \end{array} \right\},$$

where the randomness of the experiment comes from the encoding algorithm. We say a coding scheme $(\textsc{enc}, \textsc{dec})$ *is non-malleable with respect to* $\mathcal{F}$ *if for each* $f \in \mathcal{F}$, *there exists a* PPT *simulator* $\mathcal{S}$ *such that for any message* $m \in \Sigma$, *we have*

$$\mathbf{Tamper}_m^f \approx \mathbf{Ideal}_{\mathcal{S},m} \stackrel{\text{def}}{=} \left\{ \begin{array}{c} \tilde{m} \cup \{\mathsf{same}^*\} \leftarrow \mathcal{S}^{f(\cdot)}. \\ Output: m \text{ if that is } \mathsf{same}^*; \text{ otherwise } \tilde{m}. \end{array} \right\}$$

Here the indistinguishability can be either computational or statistical.

We can extend the notion of non-malleability to leakage resilience (simultaneously) as the work of Liu and Lysyanskaya [44].

Definition 2.3 (Non-malleability and Leakage Resilience [44])**.** *Let* k *be the security parameter,* $\mathcal{F}$, $\mathcal{G}$ *be some families of functions. For each function* $f \in \mathcal{F}$, $g \in \mathcal{G}$, *and* $m \in \Sigma$, *define the tamper-leak experiment:*

$$\mathbf{TamperLeak}_m^{f,g} \stackrel{\text{def}}{=} \left\{ \begin{array}{c} c \leftarrow \textsc{enc}(m), \tilde{c} := f(c), \tilde{m} := \textsc{dec}(\tilde{c}). \\ Output: \ (\tilde{m}, g(c)). \end{array} \right\},$$

where the randomness of the experiment comes from the encoding algorithm. We say a coding scheme $(\textsc{enc}, \textsc{dec})$ *is non-malleable and leakage resilience with respect to* $\mathcal{F}$ *and* $\mathcal{G}$ *if for any* $f \in \mathcal{F}$, $g \in \mathcal{G}$, *there exists a* PPT *simulator* $\mathcal{S}$ *such that for any message* $m \in \Sigma$, *we have*

$$\mathbf{TamperLeak}_m^{f,g} \approx \mathbf{Ideal}_{\mathcal{S},m} \stackrel{\text{def}}{=} \left\{ \begin{array}{c} (\tilde{m} \cup \{\mathsf{same}^*\}, \ell) \leftarrow \mathcal{S}^{f(\cdot),g(\cdot)}. \\ Output: (m, \ell) \text{ if that is } \mathsf{same}^*; \text{ otherwise } (\tilde{m}, \ell). \end{array} \right\}$$

Here the indistinguishability can be either computational or statistical.

2.2 New Definitions – Codes with Local Properties

In this section, we consider coding schemes with extra *local* properties – decodability and updatability. Intuitively, this gives a way to encode blocks of messages, such that in order to decode (retrieve) a single block of the messages, one only needs to read a small number of blocks of the codeword; similarly, in order to update a single block of the messages, one only needs to update a few blocks of the codeword.

Definition 2.4 (Locally Decodable and Updatable Code)**.** *Let* $\Sigma, \hat{\Sigma}$ *be sets of strings, and* $n, \hat{n}, p, q$ *be some parameters. An* $(n, \hat{n}, p, q)$ *locally decodable and updatable coding scheme consists of three algorithms* $(\textsc{enc}, \textsc{dec}, \textsc{update})$ *with the following syntax:*

- *The encoding algorithm* ENC *(*perhaps randomized*) takes input an n-block (in Σ) message and outputs an $\hat{n}$-block (in $\hat{\Sigma}$) codeword.*
- *The (local) decoding algorithm* DEC *takes input an index in $[n]$, reads at most p blocks of the codeword, and outputs a block of message in Σ. The overall decoding algorithm simply outputs* $(\text{DEC}(1), \text{DEC}(2), \ldots, \text{DEC}(n))$.
- *The (local) updating algorithm* UPDATE *(*perhaps randomized*) takes inputs an index in $[n]$ and a string in $\Sigma \cup \{\epsilon\}$, and reads/writes at most q blocks of the codeword. Here the string ϵ denotes the procedure of refreshing without changing anything.*

Let $C \in \hat{\Sigma}^{\hat{n}}$ be a codeword. For convenience, we denote $\text{DEC}^C, \text{UPDATE}^C$ as the processes of reading/writing individual block of the codeword, i.e. the codeword oracle returns or modifies individual block upon a query. Here we view C as a random access memory where the algorithms can read/write to the memory C at individual different locations.

Remark 2.5. *Throughout this paper, we only consider non-adaptive decoding and updating, which means the algorithms* DEC *and* UPDATE *compute all their queries at the same time before seeing the answers, and the computation only depends on the input i (the location). In contrast, an adaptive algorithm can compute a query based on the answer from previous queries. After learning the answer to such query, then it can make another query. We leave it as an interesting open question to construct more efficient schemes using adaptive queries.*

Then we define the requirements of the coding scheme.

Definition 2.6 (Correctness). *An $(n, \hat{n}, p, q)$ locally decodable and updatable coding scheme (with respect to $\Sigma, \hat{\Sigma}$) satisfies the following properties. For any message $M = (m_1, m_2, \ldots, m_n) \in \Sigma^n$, let $C = (c_1, c_2, \ldots, c_{\hat{n}}) \leftarrow \text{ENC}(M)$ be a codeword output by the encoding algorithm. Then we have:*
- *for any index $i \in [n]$, $\Pr[\text{DEC}^C(i) = m_i] = 1$, where the probability is over the randomness of the encoding algorithm.*
- *for any update procedure with input $(j, m') \in [n] \times \Sigma \cup \{\epsilon\}$, let C' be the resulting codeword by running $\text{UPDATE}^C(j, m')$. Then we have $\Pr[\text{DEC}^{C'}(j) = m'] = 1$, where the probability is over the encoding and update procedures. Moreover, the decodings of the other positions remain unchanged.*

Remark 2.7. *The correctness definition can be directly extended to handle any sequence of updates.*

Next, we define several flavors of security about non-malleability and leakage resilience.

One-time Non-malleability. First we consider one-time non-malleability of locally decodable codes, i.e., the adversary only tampers with the codeword once. This extends the idea of the non-malleable codes (as in Definition 2.2). As discussed in the introduction, we present the following definition to capture the idea that the tampering function can only do either of the following cases:
- It destroys a block (or blocks) of the underlying messages while keeping the other blocks unchanged, or
- If it modifies a block of the underlying messages to some unrelated string, then it must have modified all blocks of the underlying messages to encodings of unrelated messages.

Definition 2.8 (Non-malleability of Locally Decodable Codes)**.** *An $(n, \hat{n}, p, q)$-locally decodable coding scheme with respect to $\Sigma, \hat{\Sigma}$ is non-malleable against the tampering function class $\mathcal{F}$ if for all $f \in \mathcal{F}$, there exists some simulator $\mathcal{S}$ such that for any $M = (m_1, \ldots, m_n) \in \Sigma^n$, the experiment $\mathbf{Tamper}_M^f$ is (computationally) indistinguishable to the following ideal experiment $\mathbf{Ideal}_{\mathcal{S},M}$:*

- *$(\mathcal{I}, \vec{m}^*) \leftarrow \mathcal{S}(1^k)$, where $\mathcal{I} \subseteq [n]$, $\vec{m}' \in \Sigma^n$. (Intuitively $\mathcal{I}$ means the coordinates of the underlying message that have been tampered with).*
- *If $\mathcal{I} = [n]$, define $\vec{m} = \vec{m}^*$; otherwise set $\vec{m}|_{\mathcal{I}} = \perp, \vec{m}|_{\bar{\mathcal{I}}} = M|_{\bar{\mathcal{I}}}$, where $\vec{x}|_{\mathcal{I}}$ denotes the coordinates $\vec{x}[v]$ where $v \in \mathcal{I}$, and the bar denotes the complement of a set.*
- *The experiment outputs $\vec{m}$.*

Remark 2.9. *Here we make two remarks about the definition:*

1. *In the one-time security definition, we do not consider the update procedure. In the next paragraph when we define continual attacks, we will handle the update procedure explicitly.*
2. *One-time leakage resilience of locally decodable codes can be defined in the same way as Definition 2.3.*

Security against Continual Attacks. In the following, we extend the security to handle continual attacks. Here we consider a third party called *updater*, who can read the underlying messages and decide how to update the codeword. Our model allows the adversary to learn the location that the updater updated the messages, so we also allow the simulator to learn this information. This is without loss of generality if the leakage class $\mathcal{G}$ allows it, i.e. the adversary can query some $g \in \mathcal{G}$ to figure out what location was modified. On the other hand, the updater does not tell the adversary what content was encoded of the updated messages, so the simulator needs to simulate the view without such information. We can think of the updater as an honest user interacting with the codeword (read/write). The security intuitively means that even if the adversary can launch tampering and leakage attacks when the updater is interacting with the codeword, the adversary cannot learn anything about the underlying encoded messages (or the updated messages during the interaction).

Our continual experiment consists of rounds: in each round the adversary can tamper with the codeword and get partial information. At the end of each round, the updater will run UPDATE, and the codeword will be somewhat updated and refreshed. We note that if there is no refreshing procedure, then no coding scheme can be secure against continual leakage attack even for one-bit leakage at a time[3], so this property is necessary. Our concept of "continuity" is different from that of Faust et al. [24], who considered continual attacks on the same original codeword (the tampering functions can be chosen adaptively). Our model does not allow this type of "resetting attacks." Once a codeword has been modified to $f(C)$, the next tampering function will be applied on $f(C)$.

We remark that the one-time security can be easily extended to the continual case (using a standard hybrid argument) if the update procedure re-encodes the *whole* underlying messages (c.f. see the results in the work [44]). However, in the setting above, we emphasize on the *local property*, so this approach does not work. How to do a local update while maintaining tamper and leakage resilience makes the continual case challenging!

[3]If there is no refreshing procedure, then the adversary can eventually learn the whole codeword bit-by-bit by leakage. Thus he can learn the underlying message.

Definition 2.10 (Continual Tampering and Leakage Experiment)**.** *Let k be the security parameter, $\mathcal{F}, \mathcal{G}$ be some families of functions. Let* $(\text{ENC}, \text{DEC}, \text{UPDATE})$ *be an $(n, \hat{n}, p, q)$-locally decodable and updatable coding scheme with respect to $\Sigma, \hat{\Sigma}$. Let $\mathcal{U}$ be an updater that takes input a message $M \in \Sigma^n$ and outputs an index $i \in [n]$ and $m \in \Sigma$. Then for any blocks of messages $M = (m_1, m_2, \ldots, m_n) \in \Sigma^n$, and any (non-uniform) adversary $\mathcal{A}$, any updater $\mathcal{U}$, define the following continual experiment* $\mathbf{CTamperLeak}_{\mathcal{A},\mathcal{U},M}$*:*

- *The challenger first computes an initial encoding $C^{(1)} \leftarrow \text{ENC}(M)$.*
- *Then the following procedure repeats, at each round j, let $C^{(j)}$ be the current codeword and $M^{(j)}$ be the underlying message:*
  - *$\mathcal{A}$ sends either a tampering function $f \in \mathcal{F}$ and/or a leakage function $g \in \mathcal{G}$ to the challenger.*
  - *The challenger replaces the codeword with $f(C^{(j)})$, or sends back a leakage $\ell^{(j)} = g(C^{(j)})$.*
  - *We define $\vec{m}^{(j)} \stackrel{\text{def}}{=} \left(\text{DEC}^{f(C^{(j)})}(1), \ldots, \text{DEC}^{f(C^{(j)})}(n)\right)$.*
  - *Then the updater computes $(i^{(j)}, m) \leftarrow \mathcal{U}(\vec{m}^{(j)})$ for the challenger.*
  - *Then the challenger runs $\text{UPDATE}^{f(C^{(j)})}(i^{(j)}, m)$ and sends the index $i^{(j)}$ to $\mathcal{A}$.*
  - *$\mathcal{A}$ may terminate the procedure at any point.*
- *Let t be the total number of rounds above. At the end, the experiment outputs*

$$\left(\ell^{(1)}, \ell^{(2)}, \ldots, \ell^{(t)}, \vec{m}^{(1)}, \ldots, \vec{m}^{(t)}, i^{(1)}, \ldots, i^{(t)}\right).$$

Definition 2.11 (Non-malleability and Leakage Resilience against Continual Attacks)**.** *An $(n, \hat{n}, p, q)$-locally decodable and updatable coding scheme with respect to $\Sigma, \hat{\Sigma}$ is continual non-malleable against $\mathcal{F}$ and leakage resilient against $\mathcal{G}$ if for all* PPT *(non-uniform) adversaries $\mathcal{A}$, and* PPT *updaters $\mathcal{U}$, there exists some* PPT *(non-uniform) simulator $\mathcal{S}$ such that for any $M = (m_1, \ldots, m_n) \in \Sigma^n$,* $\mathbf{CTamperLeak}_{\mathcal{A},\mathcal{U},M}$ *is (computationally) indistinguishable to the following ideal experiment* $\mathbf{Ideal}_{\mathcal{S},\mathcal{U},M}$*:*

- *The experiment proceeds in rounds. Let $M^{(1)} = M$ be the initial message.*
- *At each round j, the experiment runs the following procedure:*
  - *At the beginning of each round, $\mathcal{S}$ outputs $(\ell^{(j)}, \mathcal{I}^{(j)}, \vec{w}^{(j)})$, where $\mathcal{I}^{(j)} \subseteq [n]$.*
  - *Define*
  $$\vec{m}^{(j)} = \begin{cases} \vec{w}^{(j)} & \text{if } \mathcal{I}^{(j)} = [n] \\ \vec{m}^{(j)}|_{\mathcal{I}^{(j)}} := \perp, \vec{m}^{(j)}|_{\bar{\mathcal{I}}^{(j)}} := M^{(j)}|_{\bar{\mathcal{I}}^{(j)}} & \text{otherwise,} \end{cases}$$
  where $\vec{x}|_{\mathcal{I}}$ denotes the coordinates $\vec{x}[v]$ where $v \in \mathcal{I}$, and the bar denotes the complement of a set.
  - *The updater runs $(i^{(j)}, m) \leftarrow \mathcal{U}(\vec{m}^{(j)})$ and sends the index $i^{(j)}$ to the simulator. Then the experiment updates $M^{(j+1)}$ as follows: set $M^{(j+1)} := M^{(j)}$ for all coordinates except $i^{(j)}$, and set $M^{(j+1)}[i^{(j)}] := m$.*
- *Let t be the total number of rounds above. At the end, the experiment outputs*

$$\left(\ell^{(1)}, \ell^{(2)}, \ldots, \ell^{(t)}, \vec{m}^{(1)}, \ldots, \vec{m}^{(t)}, i^{(1)}, \ldots, i^{(t)}\right).$$

3 Our Constructions

In this section, we present two constructions. As a warm-up, we first present a construction that is one-time secure to demonstrate the idea of achieving non-malleability, local decodability and updatability simultaneously. Then in the next section, we show how to make the construction secure against continual attacks.

3.1 Preliminary: Symmetric Encryption

A symmetric encryption scheme consists of three PPT algorithms $(\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ such that:

- The key generation algorithm Gen takes as input a security parameter 1^k returns a key sk.
- The encryption algorithm $\mathsf{Encrypt}$ takes as input a key sk, and a message m. It returns a ciphertext $c \leftarrow \mathsf{Encrypt}_{\mathsf{sk}}(m)$.
- The decryption algorithm $\mathsf{Decrypt}$ takes as input a secret key sk, and a ciphertext c. It returns a message m or a distinguished symbol $\perp$. We write this as $m = \mathsf{Decrypt}_{\mathsf{sk}}(c)$

We require that for any m in the message space, it should hold that

$$\Pr[\mathsf{sk} \leftarrow \mathsf{Gen}(1^k);\ \mathsf{Decrypt}_{\mathsf{sk}}(\mathsf{Encrypt}_{\mathsf{sk}}(m)) = m] = 1.$$

We next define semantical security, and then the authenticity. In the following, we define a left-or-right encryption oracle $\mathsf{LR}_{\mathsf{sk},b}(\cdot,\cdot)$ with $b \in \{0,1\}$ and $|m_0| = |m_1|$ as follows:

$$\mathsf{LR}_{\mathsf{sk},b}(m_0, m_1) \stackrel{\text{def}}{=} \mathsf{Encrypt}_{\mathsf{sk}}(m_b).$$

Definition 3.1 (Semantical Security)**.** *A symmetric encryption scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ is semantically secure if for any non-uniform* PPT *adversary $\mathcal{A}$, it holds that $|2\cdot\mathbf{Adv}_{\mathcal{E}}^{\text{priv}}(\mathcal{A}) - 1| = \mathsf{negl}(k)$ where*

$$\mathbf{Adv}_{\mathcal{E}}^{\text{priv}}(\mathcal{A}) = \Pr\left[\mathsf{sk} \leftarrow \mathsf{Gen}(1^k); b \leftarrow \{0,1\} : \mathcal{A}^{\mathsf{LR}_{\mathsf{sk},b}(\cdot,\cdot)}(1^k) = b\right].$$

Definition 3.2 (Authenticity [39, 5, 4])**.** *A symmetric encryption scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ is semantically secure if for any non-uniform* PPT *adversary $\mathcal{A}$, it holds that $\mathbf{Adv}_{\mathcal{E}}^{\text{auth}}(\mathcal{A}) = \mathsf{negl}(k)$ where*

$$\mathbf{Adv}_{\mathcal{E}}^{\text{auth}}(\mathcal{A}) = \Pr[\mathsf{sk} \leftarrow \mathsf{Gen}(1^k), c^* \leftarrow \mathcal{A}^{\mathsf{Encrypt}_{\mathsf{sk}}(\cdot)} : c^* \notin \mathsf{Q} \ \wedge \ \mathsf{Decrypt}_{\mathsf{sk}}(c^*) \notin \perp]$$

where Q is the query history $\mathcal{A}$ made to the encryption oracle.

3.2 A First Attempt – One-time Security

Construction. Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ be a symmetric encryption scheme, $\mathsf{NMC} = (\textsc{enc}, \textsc{dec})$ be a coding scheme. Then we consider the following coding scheme:

- $\textsc{enc}(M)$: on input $M = (m_1, m_2, \ldots, m_n)$, the algorithm first generates the encryption key $\mathsf{sk} \leftarrow \mathcal{E}.\mathsf{Gen}(1^k)$. Then it computes $c \leftarrow \mathsf{NMC}.\textsc{enc}(\mathsf{sk})$, $e_i \leftarrow \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m_i, i)$ for $i \in [n]$. The algorithm finally outputs a codeword $C = (c, e_1, e_2, \ldots, e_n)$.
- $\textsc{dec}^C(i)$: on input $i \in [n]$, the algorithm reads the first block and the $(i+1)$-st block of the codeword to retrieve (c, e_i). Then it runs $\mathsf{sk} := \mathsf{NMC}.\textsc{dec}(c)$. If the decoding algorithm outputs $\perp$, then it outputs $\perp$ and terminates. Else, it computes $(m_i, i^*) = \mathcal{E}.\mathsf{Decrypt}_{\mathsf{sk}}(e_i)$. If $i^* \neq i$, or the decryption fails, the algorithm outputs $\perp$. If all the above verifications pass, the algorithm outputs m_i.

- $\textsc{update}(i, m')$: on inputs an index $i \in [n]$, a block of message $m' \in \Sigma$, the algorithm runs $\textsc{dec}^C(i)$ to retrieve (c, e_i) and (sk, m_i, i). If the decoding algorithm returns $\perp$, the algorithm writes $\perp$ to the first block and the $(i+1)$-st block. Otherwise, it computes a fresh encoding $c' \leftarrow \mathsf{NMC}.\textsc{enc}(\mathsf{sk})$, and a fresh ciphertext $e'_i \leftarrow \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m', i)$. Then it writes back the first block and the $(i+1)$-st block with (c', e'_i).

To analyze the coding scheme, we make the following assumptions of the parameters in the underlying scheme for convenience:

1. The size of the encryption key is k (security parameter), i.e. $|\mathsf{sk}| = k$.
2. Let Σ be a set, and the encryption scheme supports messages of length $|\Sigma| + \log n$. The ciphertexts are in the space $\hat{\Sigma}$.
3. The length of $|\mathsf{NMC}.\textsc{enc}(\mathsf{sk})|$ is less than $|\hat{\Sigma}|$.

Then clearly, the above coding scheme is an $(n, n+1, 2, 2)$-locally updatable and decodable code with respect to $\Sigma, \hat{\Sigma}$. The correctness of the scheme is obvious by inspection. The rate (ratio of the length of messages to that of codewords) of the coding scheme is $1 - o(1)$.

Theorem 3.3. *Assume $\mathcal{E}$ is a symmetric authenticated encryption scheme, and NMC is a non-malleable code against the tampering function class $\mathcal{F}$. Then the coding scheme presented above is one-time non-malleable against the tampering class*

$$\bar{\mathcal{F}} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} f : \hat{\Sigma}^{n+1} \to \hat{\Sigma}^{n+1} \textit{ and } |f| \le \mathrm{poly}(k), \textit{ such that :} \\ f = (f_1, f_2),\ f_1 : \hat{\Sigma}^{n+1} \to \hat{\Sigma},\ f_2 : \hat{\Sigma}^{n} \to \hat{\Sigma}^{n}, \\ \forall (x_2, \ldots, x_{n+1}) \in \hat{\Sigma}^n, f_1(\cdot, x_2, \ldots, x_{n+1}) \in \mathcal{F} \\ f(x_1, x_2, \ldots, x_{n+1}) = (f_1(x_1, x_2, \ldots, x_{n+1}), f_2(x_2, \ldots, x_{n+1}))\,. \end{array} \right\}.$$

We have presented the intuition in the introduction. Before giving the detailed proof, we make the following remark.

Remark 3.4. *The function class $\bar{\mathcal{F}}$ may look complex, yet the intuition is simple. The tampering function restricted in the first block (the underlying non-malleable code) falls into the class $\mathcal{F}$ – this is captured by $f_1 \in \mathcal{F}$; on the other hand, we just require the function restricted in the rest of the blocks to be polynomial-sized – this is captured by $|f_2| \le |f| \le \mathrm{poly}(k)$.*

For our construction, it is inherent that the function f_2 cannot depend on x_1 arbitrarily. Suppose this is not the case, then f_2 can first decode the non-malleable code, encrypt the decoded value and write the ciphertext into x_2, which breaks non-malleability. However, if the underlying coding scheme is non-malleable and also leakage resilient to $\mathcal{G}$, then we can allow f_2 to get additional information $g(x_1)$ for any $g \in \mathcal{G}$. Moreover, the above construction is one-time leakage resilient.

We present the above simpler version for clarity of exposition, and give this remark that our construction actually achieves security against a broader class of tampering attacks.

Proof sketch of Theorem 3.3. To show the theorem, for any function $f \in \bar{\mathcal{F}}$, we need to construct a $\textsc{ppt}$ simulator $\mathcal{S}$ such that for any message blocks $M = (m_1, \ldots, m_n)$, we have $\mathbf{Tamper}^f_M \stackrel{c}{\approx} \mathbf{Ideal}_{\mathcal{S},M}$ as Definition 2.8. We describe the simulator as follows; here the $\textsc{ppt}$ simulator $\mathcal{S}$ has oracle access to $f = (f_1, f_2) \in \bar{\mathcal{F}}$.

- $\mathcal{S}^{f(\cdot)}$ first runs $\mathsf{sk} \leftarrow \mathcal{E}.\mathsf{Gen}(1^k)$ and computes n encryptions of 0, i.e. $e_i \leftarrow \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(0)$ for $i \in [n]$.
- Let $f'_1(\cdot) \stackrel{\text{def}}{=} f_1(\cdot, e_1, e_2, \ldots, e_n)$, and let $\mathcal{S}'$ be the underlying simulator of the non-malleable code NMC with respect to the tampering function f'_1. Then $\mathcal{S}^{f(\cdot)}$ simulates $\mathcal{S}'^{f'_1(\cdot)}$ internally;

here $\mathcal{S}$ uses the external oracle access to f to compute the responses for the queries made by $\mathcal{S}'$. At some point, $\mathcal{S}'$ returns an output $m' \in \Sigma \cup \{\mathsf{same}^*\}$.

- If $m' = \mathsf{same}^*$, then $\mathcal{S}$ computes $(e'_1, e'_2, \dots, e'_n) \leftarrow f_2(e_1, e_2, \dots, e_n)$. Let $\mathcal{I}$ be set of the indices where e' is not equal to e, i.e., $\mathcal{I} = \{i : e'_i \neq e_i\}$. Then $\mathcal{S}$ outputs $(\mathcal{I}, \vec{\epsilon})$, where $\vec{\epsilon}$ denotes the empty vector.
- Else if $m' \neq \mathsf{same}^*$, $\mathcal{S}$ sets $\mathsf{sk}' := m'$, and computes $(e'_1, e'_2, \dots, e'_n) \leftarrow f_2(e_1, e_2, \dots, e_n)$, and sets $\vec{m}^* := (\mathcal{E}.\mathsf{Decrypt}_{\mathsf{sk}'}(e'_1), \dots, \mathcal{E}.\mathsf{Decrypt}_{\mathsf{sk}'}(e'_n))$. Then $\mathcal{S}$ outputs $([n], \vec{m}^*)$.

To show $\mathbf{Tamper}^f_M \approx \mathbf{Ideal}_{\mathcal{S},M}$, we consider an intermediate hybrid $\mathbf{Ideal}_{\mathcal{S}^*,M}$ where $\mathcal{S}^*$ is the same as $\mathcal{S}$, except in the first place it generates $e_i \leftarrow \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m_i, i)$ for $i \in [n]$. Then we can argue that $\mathbf{Ideal}_{\mathcal{S}^*,M} \approx \mathbf{Ideal}_{\mathcal{S},M}$. This is by the property of semantic security of the encryption scheme: suppose an adversary can distinguish $\mathbf{Ideal}_{\mathcal{S}^*,M}$ from $\mathbf{Ideal}_{\mathcal{S},M}$, then we can build a reduction to break the encryption scheme by embedding the challenge ciphertexts in the first place (of $\mathcal{S}$ and $\mathcal{S}^*$), and then simulating the rest procedures, which are identical in the two cases.

Then we consider another intermediate experiment $\mathbf{Ideal}_{\mathcal{S}^{**},M}$, where the experiment is the same as $\mathbf{Ideal}_{\mathcal{S}^*,M}$ except when $\mathcal{S}^{**}$ obtains m' by running real tampering experiment $f_1(\mathsf{NMC}.\textsc{enc}(\mathsf{sk}), e_1, e_2, \dots, e_n)$ and outputs same^* if the outcome is the original sk. By the property of the underlying non-malleable code NMC, $\mathbf{Ideal}_{\mathcal{S}^*,M} \approx \mathbf{Ideal}_{\mathcal{S}^{**},M}$. Finally, we want to show that $\mathbf{Tamper}^f_M \approx \mathbf{Ideal}_{\mathcal{S}^{**},M}$. This is by the properties of the authenticity of the encryption scheme. Since with overwhelming probability, if f_2 changed any block of the ciphertext, the decryption will be $\perp$. This completes the sketch of the proof. □

3.3 Achieving Security against Continual Attacks

As discussed in the introduction, the above construction is not secure if continual tampering and leakage is allowed – the adversary can use a rewind attack to modify the underlying message to some old/related messages. We handle this challenge using a technique of Merkle tree, which preserves local properties of the above scheme. We present the construction in the following:

Definition 3.5 (Merkle Tree)**.** *Let $h : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$ be a hash function that maps two blocks of messages to one.*[4] *A Merkle Tree $\mathsf{Tree}_h(M)$ takes input a message $M = (m_1, m_2, \dots, m_n) \in \mathcal{X}^n$. Then it applies the hash on each pair (m_{2i-1}, m_{2i}), and resulting in $n/2$ blocks. Then again, it partitions the blocks into pairs and applies the hash on the pairs, which results in $n/4$ blocks. This is repeated $\log n$ times, resulting a binary tree with hash values, until one block remains. We call this value the root of Merkle Tree denoted $\mathsf{Root}_h(M)$, and the internal nodes (including the root) as $\mathsf{Tree}_h(M)$. Here M can be viewed as leaves.*

Theorem 3.6. *Assuming h is a collision resistant hash function. Then for any message $M = (m_1, m_2, \dots, m_n) \in \mathcal{X}^n$, any polynomial time adversary $\mathcal{A}$,* $\Pr\Big[(m'_i, p_i) \leftarrow \mathcal{A}(M, h) : m'_i \neq m_i, p_i$ *is a consistent path with* $\mathsf{Root}_h(M)\Big] \leq \mathsf{negl}(k)$.

Moreover, given a path p_i passing the leaf m_i, and a new value m'_i, there is an algorithm that computes $\mathsf{Root}_h(M')$ in time $\mathrm{poly}(\log n, k)$, where $M' = (m_1, \dots, m_{i-1}, m'_i, m_{i+1}, \dots, m_n)$.

Construction. Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ be a symmetric encryption scheme, $\mathsf{NMC} = (\textsc{enc}, \textsc{dec})$ be a non-malleable code, H is a family of collision resistance hash functions. Then we consider the following coding scheme:

[4] Here we assume $|\mathcal{X}|$ is greater than the security parameter.

- $\textsc{enc}(M)$: on input $M = (m_1, m_2, \ldots, m_n)$, the algorithm first generates encryption key $\mathsf{sk} \leftarrow \mathcal{E}.\mathsf{Gen}(1^k)$ and $h \leftarrow H$. Then it computes $e_i \leftarrow \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m_i)$ for $i \in [n]$, and $T = \mathsf{Tree}_h(e_1, \ldots, e_n), R = \mathsf{Root}_h(e_1, \ldots, e_n)$. Then it computes $c \leftarrow \mathsf{NMC}.\textsc{enc}(\mathsf{sk}, R, h)$, The algorithm finally outputs a codeword $C = (c, e_1, e_2, \ldots, e_n, T)$.
- $\textsc{dec}^C(i)$: on input $i \in [n]$, the algorithm reads the first block, the $(i+1)$-st block, and a path p in the tree (from the root to the leaf i), and it retrieve (c, e_i, p). Then it runs $(\mathsf{sk}, R, h) = \mathsf{NMC}.\textsc{dec}(c)$. If the decoding algorithm outputs $\perp$, or the path is not consistent with the root R, then it outputs $\perp$ and terminates. Else, it computes $m_i = \mathcal{E}.\mathsf{Decrypt}_{\mathsf{sk}}(e_i)$. If the decryption fails, output $\perp$. If all the above verifications pass, the algorithm outputs m_i.
- $\textsc{update}(i, m')$: on inputs an index $i \in [n]$, a block of message $m' \in \Sigma$, the algorithm runs $\textsc{dec}^C(i)$ to retrieve (c, e_i, p). Then the algorithm can derive $(\mathsf{sk}, R, h) = \mathsf{NMC}.\textsc{dec}(c)$. If the decoding algorithm returns $\perp$, the update writes $\perp$ to the first block, which denotes failure. Otherwise, it computes a fresh ciphertext $e'_i \leftarrow \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(m')$, a new path p' (that replaces e_i by e'_i) and a new root R', which is consistent with the new leaf value e'_i. (Note that this can be done given only the old path p as Theorem 3.6.) Finally, it computes a fresh encoding $c' \leftarrow \mathsf{NMC}.\textsc{enc}(\mathsf{sk}, R', h)$. Then it writes back the first block, the $(i+1)$-st block, and the new path blocks with (c', e'_i, p').

To analyze the coding scheme, we make the following assumptions of the parameters in the underlying scheme for convenience:

1. The size of the encryption key is k (security parameter), i.e. $|\mathsf{sk}| = k$ and the length of the output of the hash function is k.
2. Let Σ be a set, and the encryption scheme supports messages of length $|\Sigma|$. The ciphertexts are in the space $\hat{\Sigma}$.
3. The length of $|\mathsf{NMC}.\textsc{enc}(\mathsf{sk}, v)|$ is less than $|\hat{\Sigma}|$, where $|v| = k$.

Clearly, the above coding scheme is an $(n, 2n+1, O(\log n), O(\log n))$-locally updatable and decodable code with respect to $\Sigma, \hat{\Sigma}$. The correctness of the scheme is obvious by inspection. The rate (ratio of the length of messages to that of codewords) of the coding scheme is $1/2 - o(1)$.

Theorem 3.7. *Assume $\mathcal{E}$ is a semantically secure symmetric encryption scheme, and NMC is a non-malleable code against the tampering function class $\mathcal{F}$, and leakage resilient against the function class $\mathcal{G}$. Then the coding scheme presented above is non-malleable against continual attacks of the tampering class*

$$\bar{\mathcal{F}} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} f : \hat{\Sigma}^{2n+1} \to \hat{\Sigma}^{2n+1} \textit{ and } |f| \leq \mathrm{poly}(k), \textit{ such that :} \\ f = (f_1, f_2),\ f_1 : \hat{\Sigma}^{2n+1} \to \hat{\Sigma},\ f_2 : \hat{\Sigma}^{2n} \to \hat{\Sigma}^{2n}, \\ \forall (x_2, \ldots, x_{2n+1}) \in \hat{\Sigma}^n, f_1(\ \cdot\ , x_2, \ldots, x_{2n+1}) \in \mathcal{F}, \\ f(x_1, x_2, \ldots, x_{2n+1}) = (f_1(x_1, x_2, \ldots, x_{2n+1}), f_2(x_2, \ldots, x_{2n+1}))\,. \end{array} \right\},$$

and is leakage resilient against the class

$$\bar{\mathcal{G}} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} g : \hat{\Sigma}^{2n+1} \to \mathcal{Y} \textit{ and } |g| \leq \mathrm{poly}(k), \textit{ such that :} \\ \forall\ (x_2, \ldots, x_{2n+1}) \in \hat{\Sigma}^n, g(\ \cdot\ , x_2, \ldots, x_{2n+1}) \in \mathcal{G}. \end{array} \right\}.$$

The intuition of this construction can be found in the introduction. Before giving the detailed proof, we make a remark.

Remark 3.8. *Actually our construction is secure against a broader class of tampering functions. The f_2 part can depend on $g'(x_1)$ as long as the function $g'(\cdot)$ together with the leakage* *function $g(\cdot, x_2, \ldots, x_{2n+1})$ belong to $\mathcal{G}$. That is, the tampering function $f = (f_1, f_2, g')$ and the leakage function g satisfy the constraint $g'(\cdot) \circ g(\cdot, x_2, \ldots, x_{2n+1}) \in \mathcal{G}$ (Here we use $\circ$ to denote concatenation). For presentation clarity, we choose to describe the simpler but slightly smaller class of functions.*

Proof of Theorem 3.7. To prove the theorem, for any adversary $\mathcal{A}$, we need to construct a simulator $\mathcal{S}$, such that for initial message $M \in \Sigma^n$, any updater $\mathcal{U}$, the experiment of continual attacks $\textbf{TamperLeak}_{\mathcal{A},\mathcal{U},M}$ is indistinguishable from the ideal experiment $\textbf{Ideal}_{\mathcal{S},\mathcal{U},M}$.

The simulator $\mathcal{S}$ first samples random coins for the updater $\mathcal{U}$, so its output just depends on its input given the random coins. Then $\mathcal{S}$ works as follows:

- Initially $\mathcal{S}$ samples $\mathsf{sk} \leftarrow \mathcal{E}.\mathsf{Gen}(1^k), h \leftarrow H$, and then generates n encryptions of 0, i.e., $\vec{e}^{(1)} := (e_1, e_2, \ldots, e_n)$ where $e_i \leftarrow \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(0)$ for $i \in [n]$. Then $\mathcal{S}$ computes $T^{(1)} := \mathsf{Tree}_h(e_1, \ldots, e_n)$. Here let $R^{(1)}$ be the root of the tree. $\mathcal{S}$ keeps global variables: sk, h, a flag $\mathsf{flag} = 0$, and a string $C = \epsilon$ (empty string).
- At each round j, let $g^{(j)} \in \bar{\mathcal{G}}, f^{(j)} = (f_1^{(j)}, f_2^{(j)}) \in \bar{\mathcal{F}}$ be some leakage/ tampering functions specified by the adversary. If the flag is 0, i.e. $\mathsf{flag} = 0$, then $\mathcal{S}$ does the following:
  - First, $\mathcal{S}$ sets $(e_1, e_2, \ldots, e_n) := \vec{e}^{(j)}$, $T := T^{(j)}$, and $R := R^{(j)}$. Then $\mathcal{S}$ defines $f_1'(\cdot) \stackrel{\text{def}}{=} f_1^{(j)}(\cdot, e_1, e_2, \ldots, e_n, T)$, and $g'(\cdot) \stackrel{\text{def}}{=} g^{(j)}(\cdot, e_1, e_2, \ldots, e_n, T)$. Let $\mathcal{S}'$ be the simulator of the underlying leakage resilient non-malleable code NMC with respect to the tampering and leakage functions $f_1'(\cdot)$ and $g'(\cdot)$.
  - Then $\mathcal{S}$ computes $(m', \ell') \leftarrow \mathcal{S}'^{f_1'(\cdot), g'(\cdot)}$, and sets $\ell^{(j)} := \ell'$, and $(e_1', e_2', \ldots, e_n', T') := f_2^{(j)}(e_1, e_2, \ldots, e_n, T)$.
  - If $m' = \mathsf{same}^*$, $\mathcal{S}$ sets $\mathcal{I}^{(j)} = \{u : e_u' \neq e_u\}$, i.e. the indices where e' is not equal to e, and set $\vec{w}^{(j)} := \vec{\epsilon}$, the empty vector. $\mathcal{S}$ outputs $\{\ell^{(j)}, \mathcal{I}^{(j)}, \vec{w}^{(j)}\}$ for this round.

    Then upon receiving an index $i^{(j)} \in [n]$ from the updater, then $\mathcal{S}$ checks whether the path passing the leaf $e'_{i^{(j)}}$ in the Merkle Tree T' is consistent with the root R, and does the following:
    - If the check fails, he sets $\mathsf{flag} := 1$, $C := (\perp, e_1', \ldots, e_n', T')$, and then exits the loop of this round.
    - Otherwise, he sets $\vec{e}^{(j+1)} := (e_1', e_2', \ldots, e_n')$ for all indices except $i^{(j)}$. He creates a fresh ciphertext $e \leftarrow \mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(0)$, and sets $\vec{e}^{(j+1)}[i^{(j)}] := e$ (simulating the update). He updates the path passing through the $i^{(j)}$-th leaf in T' and the root R, and set $T^{(j+1)} := T'$, $R^{(j+1)} := R$ (the updated ones).
  - Else if $m' \neq \mathsf{same}^*$, then $\mathcal{S}$ sets $\mathcal{I}^{(j)} := [n]$, and sets the flag to be 1, i.e. $\mathsf{flag} := 1$. He parses $m' = (\mathsf{sk}', h', R')$, and uses the key sk' to compute $\vec{w}^{(j)} = (\mathcal{E}.\mathsf{Decrypt}_{\mathsf{sk}'}(e_1'), \ldots, \mathcal{E}.\mathsf{Decrypt}_{\mathsf{sk}'}(e_n'))$. Then he outputs $\{\ell^{(j)}, \mathcal{I}^{(j)}, \vec{w}^{(j)}\}$ for this round.

    Then $\mathcal{S}$ computes $(i^{(j)}, m) \leftarrow \mathcal{U}(\vec{w}^{(j)})$ on his own. Let $C' = (\mathsf{NMC}.\textsc{enc}(\mathsf{sk}', h', R'), e_1', \ldots, e_n', T')$ be a codeword, and $\mathcal{S}$ runs $\textsc{update}^{C'}(i^{(j)}, m)$. Let C^* be the resulting codeword, and $\mathcal{S}$ updates the global variable $C := C^*$.
- Else if $\mathsf{flag} = 1$, $\mathcal{S}$ simulates the real experiment faithfully:
  - $\mathcal{S}$ outputs $\ell^{(j)} = g^{(j)}(C)$, and computes $C' = f^{(j)}(C)$.
  - Set $\vec{w}^{(j)}[v] := \textsc{dec}^{(C')}(v)$, i.e. running the real decoding algorithm. Then $\mathcal{S}$ outputs $\{\ell^{(j)}, \mathcal{I}^{(j)} = [n], \vec{w}^{(j)}\}$ for this round.
  - Then $\mathcal{S}$ computes $(i^{(j)}, m) \leftarrow \mathcal{U}(\vec{w}^{(j)})$ on his own, and runs $\textsc{update}^{C'}(i^{(j)}, m)$. Let C^* be the resulting codeword after the update, and $\mathcal{S}$ updates the variable $C := C^*$.

To show $\mathbf{CTamperLeak}_{\mathcal{A},\mathcal{U},M} \approx \mathbf{Ideal}_{\mathcal{S},\mathcal{U},M}$, we consider several intermediate hybrids.

Hybrid H_0: This is exactly the experiment $\mathbf{Ideal}_{\mathcal{S},\mathcal{U},M}$.

Hybrid H_1: This experiment is the same as H_0 except the simulator does not generate sk of the encryption scheme. Whenever he needs to produce a ciphertext (only in the case when $\mathsf{flag} = 0$), the hybrid provides oracle access to the encryption algorithm $\mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(\cdot)$, where the experiment samples sk privately.

It is not hard to see that the experiment H_0 is identical to H_1. Then we define another hybrid:

Hybrid H_2: This experiment is the same as H_1 except, the encryption oracle does not gives $\mathcal{E}.\mathsf{Encrypt}_{\mathsf{sk}}(0)$ to the simulator; instead, it gives encryptions of the real messages (in the first place, and in the update when $\mathsf{flag} = 0$), as in the real experiment.

By a simple reduction argument, we can establish the following claim.

Claim 3.9. *Suppose the encryption scheme is semantically secure, then H_1 is computationally indistinguishable from H_2.*

Hybrid H_3: This experiment is the same as H_2 except, the simulator does not use the underlying $\mathcal{S}'$ of the non-malleable code to produce (m', ℓ') (in the case when $\mathsf{flag} = 0$). Let R be the current root of the Merkle Tree, h be the hash function, sk be the secret key of the encryption oracle. In this experiment, the simulator generates an encoding of $\mathsf{NMC}.\textsc{enc}(\mathsf{sk}, h, R)$ and then applies the tampering and leakage function faithfully as the real experiment $\mathbf{TamperLeak}^{f,g}$. If the outcome is still (sk, h, R), then the simulator treats this as same^*. Otherwise, it uses the decoded value to proceed. Then the rest follows exactly as H_2.

Then we can establish the following claim:

Claim 3.10. *Suppose the underlying coding scheme* NMC *is non-malleable and leakage resilience against $\mathcal{F}$ and $\mathcal{G}$, then H_2 is computationally indistinguishable from H_3.*

Proof Sketch. We can show this by considering the following sub-hybrids: $H_{2,j}$: in the first j rounds, the simulator generates (m', ℓ') according to the experiment **TamperLeak**, and in the rest $\mathcal{S}'$. By the property of the coding scheme, we can show each adjacent sub-hybrids are computationally indistinguishable by reduction. Note that the simulator refreshes the encoding of (sk, h, R) at each round, so we can apply the hybrid argument. From the description, we have $H_2 = H_{2,0}$, and $H_{2,t} = H_3$, where t is the total number of rounds. □

Finally we want to show the following claim:

Claim 3.11. *Suppose the hash function comes from a collision resilient hash family, then H_3 is computationally indistinguishable from* $\mathbf{CTamperLeak}_{\mathcal{A},\mathcal{U},M}$.

Proof. We observe that the only difference between H_3 and $\mathbf{CTamperLeak}_{\mathcal{A},\mathcal{U},M}$ is the generation of $\vec{m}^{(j)}$ at each round (when the flag is 0). In the experiment $\mathbf{CTamperLeak}_{\mathcal{A},\mathcal{U},M}$, $\vec{m}^{(j)}$ is generated by honestly decoding the codeword at each position, i.e. $\left(\textsc{dec}^{f(C^{(j)})}(1), \ldots, \textsc{dec}^{f(C^{(j)})}(n)\right)$. In H_3, $\vec{m}^{(j)}$ is generated by first computing $(m', e'_1, \ldots, e'_n, T') := f(C^{(j)})$. In the case where $m' \neq \mathsf{same}^*$, the two experiments are identical. In the case where $m' = \mathsf{same}^*$, H_3 sets $\vec{m}^{(j)}[v] = \perp$ if $e'_v \neq e_v$. The only situation that these two hybrids deviate is when $e'_v \neq e_v$

but there is another consistent path in T' with the root R. For this situation, $\textsc{dec}(v) \neq \perp$ in **CTamperLeak**, but H_3 will set $\vec{m}[v] := \perp$. However, we claim this event can happen with a negligible probability, or otherwise we can break the security of the Merkle Tree (Theorem 3.6) by simulating the hybrid H_3. This completes the proof of the claim.

□

Putting everything together, we show that $\mathbf{CTamperLeak}_{\mathcal{A},\mathcal{U},M} \approx \mathbf{Ideal}_{\mathcal{S},\mathcal{U},M}$.

□

3.4 Instantiations

In this section, we describe several constructions of non-malleable codes against different classes of tampering/leakage functions. To our knowledge, we can use the explicit constructions (of the non-malleable codes) in the work [23, 11, 44, 24, 26, 1, 3].

First we overview different classes of tampering/leakage function allowed for these results: the constructions of [23] work for bit-wise tampering functions, and split-state functions in the random oracle model. The construction of Choi et al. [11] works for small block tampering functions. The construction of Liu and Lysyanskaya [44] achieves both tamper and leakage resilience against split-state functions in the common reference string (CRS) model. The construction of Dziembowski et al. [21] achieves information theoretic security against split-state tampering functions, but their scheme can only support encoding for bits, so it cannot be used in our construction. The subsequent construction by Aggarwal et al. [1] achieves information theoretic security against split-state tampering without CRS. We believe that their construction also achieves leakage resilience against some length bounded split-state leakage yet their paper did not claim it. The construction by Faust et al. [26] is non-malleable against small-sized tampering functions. Another construction by Faust et al. [24] achieves both tamper and leakage resilience in the split-state model with CRS. The construction of Aggarwal et al. [3] is non-malleable against permutation functions.

Then we remark that actually there are other non-explicit constructions: Cheraghchi and Guruswami [10] showed the relation non-malleable codes and non-malleable two source extractors (but constructing a non-malleable two-source extractor is still open), and in another work Cheraghchi and Guruswami [9] showed the existence of high rate non-malleable codes in the split-state model but did not give an explicit (efficient) construction.

Finally, we give a concrete example of what the resulting class looks like using the construction of Liu and Lysyanskaya [44] as the building block. Recall that their construction achieves both tamper and leakage resilience for split-state functions. Our construction has the form $(\mathsf{NMC}.\textsc{enc}(\mathsf{sk}, h, T), \mathsf{Encrypt}(m_1), \ldots, \mathsf{Encrypt}(m_n), T)$. So the overall leakage function g restricted in the first block (i.e. g_1) can be a (poly-sized) length-bounded split-state function; g on the other hand, can leak all the other parts. For the tampering, the overall tampering function f restricted in the first block (i.e. f_1) can be any (poly-sized) split-state function. On the other hand f restricted in the rest (i.e. f_2) can be just any poly-sized function. We also remark that f_2 can depend on a split-state leakage on the first part, say g_1, as we discussed in the previous remark above.

4 Tamper and Leakage Resilient RAM

In this section, we first introduce the notations of the Random Access Machine (RAM) model of computation in presence of tampering and leakage attacks in Section 4.1. Then we define the security of tamper and leakage resilient RAM model of computation in Section 4.2, recall the building block Oblivious RAM (ORAM) in Section 4.3, and then give a construction in Section 4.4. and the security analysis in Section 4.5.

4.1 Random Access Machines

We consider RAM programs to be interactive stateful systems $\langle \Pi, \mathsf{state}, D\rangle$, where Π denotes a next instruction function, state the current state stored in registers, and D the content of memory. Upon state and an input value d, the next instruction function outputs the next instruction I and an updated state state'. The initial state of the RAM machine, state, is set to $(\mathsf{start}, *)$. For simplicity we often denote RAM program as $\langle \Pi, D\rangle$. We consider four ways of interacting with the system:

- $\mathsf{Execute}(x)$: A user can provide the system with $\mathsf{Execute}(x)$ queries, for $x \in \{0,1\}^u$, where u is the input length. Upon receiving such query, the system computes $(y, t, D') \leftarrow \langle \Pi, D\rangle(x)$, updates the state of the system to $D := D'$ and outputs (y, t), where y denotes the output of the computation and t denotes the time (or number of executed instructions). By $\mathsf{Execute}_1(x)$ we denote the first coordinate of the output of $\mathsf{Execute}(x)$.
- $\mathsf{doNext}(x)$: A user can provide the system with $\mathsf{doNext}(x)$ queries, for $x \in \{0,1\}^u$. Upon receiving such query, if $\mathsf{state} = (\mathsf{start}, *)$, set $\mathsf{state} := (\mathsf{start}, x)$, and $d := 0^r$; Here $\rho = |\mathsf{state}|$ and $r = |d|$. The system does the following until termination:

  1. Compute $(I, \mathsf{state}') = \Pi(\mathsf{state}, d)$. Set $\mathsf{state} := \mathsf{state}'$.
  2. If $I = (\mathsf{wait})$ then set $\mathsf{state} := 0^\rho$, $d := 0^r$ and terminate.
  3. If $I = (\mathsf{stop}, z)$ then set $\mathsf{state} := (\mathsf{start}, *)$, $d := 0^r$ and terminate with output z.
  4. If $I = (\mathsf{write}, v, d')$ then set $D[v] := d'$.
  5. If $I = (\mathsf{read}, v, \perp)$ then set $d := D[v]$.

  Let $I_1, \ldots, I_\ell$ be the instructions executed by $\mathsf{doNext}(x)$. All memory addresses of executed instructions are returned to the user. Specifically, for instructions I_j of the form $(\mathsf{read}, v, \perp)$ or (write, v, d'), v is returned.

- $\mathsf{Tamper}(f)$: We also consider tampering attacks against the system, modeled by $\mathsf{Tamper}(f)$ commands, for functions f. Upon receiving such command, the system sets $D := f(D)$.
- $\mathsf{Leak}(g)$: We finally consider leakage attacks against the system, modeled by $\mathsf{Leak}(g)$ commands, for functions g. Upon receiving such command, the value of $g(D)$ is returned to the user.

Remark 4.1. *A $\mathsf{doNext}(x)$ instruction groups together instructions performed by the CPU in a single clock cycle. Intuitively, a (wait) instruction indicates that a clock cycle has ended and the CPU waits for the adversary to increment the clock. In contrast, a (stop, z) instruction indicates that the entire execution has concluded with output z. In this case, the internal state is set back to the start state.*

We require that each $\mathsf{doNext}(x)$ operation performs exactly $\ell = \ell(k) = \mathrm{poly}(k)$ instructions $I_1, \ldots, I_\ell$ where: The final instruction is of the form $I_\ell = (\mathsf{stop}, \cdot)$ or $I_\ell = (\mathsf{wait})$. For fixed $\ell_1 = \ell_1(k), \ell_2 = \ell_2(k)$ such that $\ell_1 + \ell_2 = \ell - 1$, we have that the first ℓ_1 instructions are of the form $I_\ell = (\mathsf{read}, \cdot, \perp)$ and the next ℓ_2 instructions are of the form $I_\ell = (\mathsf{write}, v, d')$. We assume that ℓ, ℓ_1, ℓ_2 are implementation-specific and public. The limitations on space are meant to model the fact that the CPU has a limited number of registers and that no persistent state is kept by the CPU between clock cycles.

Remark 4.2. *We note that* $\mathsf{Execute}(x)$ *instructions are used by the ideal world adversary—who learns only the input-output behavior of the RAM machine and the run time—as well as by the real world adversary. The real world adversary may also use the more fine-grained* $\mathsf{doNext}(x)$ *instruction. We note that given access to the* $\mathsf{doNext}(x)$ *instruction, the behavior of the* $\mathsf{Execute}(x)$ *instruction may be simulated.*

Remark 4.3. *We note that our model does not explicitly allow for leakage and tampering on instructions* I*. E.g. when an instruction* $I = (\mathsf{write}, v, d')$ *is executed, we do not directly allow tampering with the values* v, d' *or leakage on* d' *(note that* v *is entirely leaked to the adversary). Nevertheless, as discussed in the introduction, since we allow full leakage on the addresses, the adversary can use the tampering and leakage attacks on the memory to capture the attacks on the instructions. We defer a rigorous treatment and analysis of such attacks to future work. In this work, for simplicity of presentation we assume these instructions are not subject to direct attacks.*

4.2 Tamper and Leakage-Resilient (TLR) RAM

A tamper and leakage resilient (TLR) RAM compiler consists of two algorithms $(\mathsf{CompMem}, \mathsf{CompNext})$, which transform a RAM program $\langle \Pi, D \rangle$ into another program $\langle \widehat{\Pi}, \widehat{D} \rangle$ as follows: On input database D, $\mathsf{CompMem}$ initializes the memory and internal state of the compiled machine, and generates the transformed database $\widehat{D}$; On input next instruction function Π, $\mathsf{CompNext}$ generates the next instruction function of the compiled machine.

Definition 4.4. *A TLR compiler* $(\mathsf{CompMem}, \mathsf{CompNext})$ *is tamper and leakage simulatable w.r.t. function families* $\mathcal{F}, \mathcal{G}$*, if for every RAM next instruction function* Π*, and for any* PPT *(non-uniform) adversary* $\mathcal{A}$ *there exists a* PPT *(non-uniform) simulator* $\mathcal{S}$ *such that for any initial database* $D \in \{0,1\}^{\mathrm{poly}(k)}$ *we have*

$$\mathbf{TamperExec}(\mathcal{A}, \mathcal{F}, \mathcal{G}, \langle \mathsf{CompNext}(\Pi), \mathsf{CompMem}(D) \rangle) \approx \mathbf{IdealExec}(\mathcal{S}, \langle \Pi, D \rangle)$$

where **TamperExec** *and* **IdealExec** *are defined as follows:*

– $\mathbf{TamperExec}(\mathcal{A}, \mathcal{F}, \mathcal{G}, \langle \mathsf{CompNext}(\Pi), \mathsf{CompMem}(D) \rangle)$*: The adversary* $\mathcal{A}$ *interacts with the system* $\langle \mathsf{CompNext}(\Pi), \mathsf{CompMem}(D) \rangle$ *for arbitrarily many rounds of interactions where, in each round:*

  1. *The adversary can "tamper" by executing a* $\mathsf{Tamper}(f)$ *command against the system, for some* $f \in \mathcal{F}$*.*
  2. *The adversary can "leak" by executing a* $\mathsf{Leak}(g)$ *command against the system, and receiving* $g(D)$ *in return.*
  3. *The adversary requests a* $\mathsf{doNext}(x)$ *command to be executed by the system. Let* $I_1, \ldots, I_\ell$ *be the instructions executed by* $\mathsf{doNext}(x)$*. If* I_ℓ *is of the form* (stop, z) *then output* z *is returned to the adversary. Moreover, all memory addresses corresponding to instructions* $I_1, \ldots, I_{\ell-1}$ *are returned to the adversary.*

  The output of the game consists of the output of the adversary $\mathcal{A}$ *at the end of the interaction, along with (1) all input-output pairs* $(x_1, y_1), (x_2, y_2), \ldots$*, (2) all responses to leakage queries* $\ell_1, \ell_2, \ldots$ *(3) all outputs of* $\mathsf{doNext}(x_1), \mathsf{doNext}(x_2), \ldots$*.*

– $\mathbf{IdealExec}(\mathcal{S}, \langle \Pi, D \rangle)$*: The simulator interacts with the system* $\langle \Pi, D \rangle$ *for arbitrarily many rounds of interaction where, in each round, it runs an* $\mathsf{Execute}(x)$ *query for some* $x \in \{0,1\}^u$ *and receives output* (y, t)*. The output of the game consists of the output of the simulator* $\mathcal{S}$ *at the end of the interaction, along with all of the execute-query inputs and outputs.*

For simplicity of exposition, we assume henceforth that the next instruction function Π to be compiled is the universal RAM next instruction function. In other words, we assume that the program to be executed is stored in the initial database D.

4.3 Preliminary: Oblivious RAM (ORAM)

An ORAM compiler ORAM consists of two algorithms $(\mathsf{oCompMem}, \mathsf{oCompNext})$, which transform a RAM program $\langle \Pi, D\rangle$ into another program $\langle \widetilde{\Pi}, \widetilde{D}\rangle$ as follows: On input database D, $\mathsf{CompMem}$ initializes the memory and internal state of the compiled machine, and generates the transformed database $\widetilde{D}$; On input next instruction function Π, $\mathsf{CompNext}$ generates the next instruction function of the compiled machine, $\widetilde{\Pi}$.

Correctness. We require the following correctness property: For every choice of security parameter k, every initial database D and every sequence of inputs $x_1, \dots, x_p$, where $p = p(k)$ is polynomial in k, we have that with probability $1 - \mathsf{negl}(k)$ over the coins of $\mathsf{oCompMem}$,

$$(\mathsf{Execute}_1(x_1), \dots, \mathsf{Execute}_1(x_p)) = \left(\widetilde{\mathsf{Execute}}_1(x_1), \dots, \widetilde{\mathsf{Execute}}_1(x_p)\right),$$

where $\mathsf{Execute}_1(x)$ denotes the first coordinate of the output of $\mathsf{Execute}(x)$ w.r.t. $\langle \Pi, D\rangle$ and $\widetilde{\mathsf{Execute}}_1(x)$ denotes the first coordinate of the output of $\mathsf{Execute}(x)$ w.r.t. $\langle \mathsf{oCompNext}(\Pi), \mathsf{oCompMem}(D)\rangle$.

Security. Let $\mathsf{ORAM} = (\mathsf{oCompMem}, \mathsf{oCompNext})$ be an ORAM complier and consider the following experiment:

Experiment $\mathbf{Expt}^{\mathrm{oram}}_{\mathcal{A}}(k, b)$:

1. The adversary $\mathcal{A}$ selects two initial databases D_0, D_1.
2. Set initial contents of memory of the RAM machine to $\widetilde{D} := \mathsf{oCompMem}(D_b)$. Set the initial state of the RAM machine to $\mathsf{state} := (\mathsf{start}, *)$.
3. The adversary $\mathcal{A}$ and the challenger participate in the following procedure for an arbitrary number of rounds:
   - For $x \in \{0,1\}^u$, $\mathcal{A}$ submits a $\mathsf{doNext}(x)$ query.
   - Execute the $\mathsf{doNext}(x)$ query w.r.t. $\langle \mathsf{oCompNext}(\Pi), \widetilde{D}\rangle$ and update the state of the system. Let $I_1, \dots, I_\ell$ be the instructions executed by the RAM machine. For each $j \in [\ell]$, if I_j is of the form $(\cdot, v_j, \cdot)$, for some v_j, output v_j to $\mathcal{A}$. Otherwise, output $v_j = \perp$. Let $\overline{v} = v_1, \dots, v_\ell$ be the output obtained by $\mathcal{A}$ in the current round.
4. Finally, the adversary outputs a guess $b' \in \{0,1\}$. The experiment evaluates to 1 iff $b' = b$.

Definition 4.5. *An ORAM construction* $\mathsf{ORAM} = (\mathsf{oCompMem}, \mathsf{oCompNext})$ *is* access-pattern hiding *if for every* PPT *adversary $\mathcal{A}$, the following probability, taken over the randomness of the experiment and $b \in \{0,1\}$, is negligible:*

$$\left| \Pr[\mathbf{Expt}^{\mathrm{oram}}_{\mathcal{A}}(k, b) = 1] - \frac{1}{2} \right|.$$

4.4 TLR-RAM Construction

Here we first give a high-level description of our construction. More detailed construction and a theorem statement follow. The security proof will be given in next section.

High-level Description of Construction Let D be the initial database and let $\mathsf{ORAM} = (\mathsf{oCompMem}, \mathsf{oCompNext})$ be an ORAM compiler. Let $\mathsf{NMCode} = (\textsc{enc}, \textsc{dec}, \textsc{update})$ be a locally decodable and updatable code. We present the following construction $\mathsf{TLR\text{-}RAM} = (\mathsf{CompMem}, \mathsf{CompNext})$ of a tamper and leakage resilient RAM compiler. In order to make our presentation more intuitive, instead of specifying the next message function $\mathsf{CompNext}(\Pi)$, we specify the pseudocode for the $\mathsf{doNext}(x)$ instruction of the compiled machine. We note that $\mathsf{CompNext}(\Pi)$ is implicitly defined by this description.

$\mathsf{TLR\text{-}RAM}$ takes as input an initial database D and a next instruction function Π and does the following:

- $\mathsf{CompMem}$: On input security parameter k and initial database D, $\mathsf{CompMem}$ does:
  - Compute $\widetilde{D} \leftarrow \mathsf{oCompMem}(D)$, and output $\widehat{D} \leftarrow \textsc{enc}(\widetilde{D})$.
  - Initialize the ORAM state $\mathsf{state}_{\mathsf{ORAM}} := (\mathsf{start}, *)$ and $d_{\mathsf{ORAM}} := 0^r$, where $r = |d_{\mathsf{ORAM}}|$.
- $\mathsf{doNext}(x)$: On input x, do the following until termination:
  1. If $d_{\mathsf{ORAM}} = \bot$ then abort.
  2. Compute $(I, \mathsf{state}'_{\mathsf{ORAM}}) \leftarrow \mathsf{oCompNext}(\Pi)(\mathsf{state}_{\mathsf{ORAM}}, d_{\mathsf{ORAM}})$. Set $\mathsf{state}_{\mathsf{ORAM}} := \mathsf{state}'_{\mathsf{ORAM}}$.
  3. If $I = (\mathsf{wait})$ then set $\mathsf{state}_{\mathsf{ORAM}} := 0^\rho$ and $d_{\mathsf{ORAM}} := 0^r$ and terminate. Here $\rho = |\mathsf{state}_{\mathsf{ORAM}}|$ and $r = |d_{\mathsf{ORAM}}|$.
  4. If $I = (\mathsf{stop}, z)$ then set $\mathsf{state}_{\mathsf{ORAM}} := (\mathsf{start}, *)$, $d := 0^r$ and terminate with output z.
  5. If $I = (\mathsf{write}, v, d')$ then run $\textsc{update}^{\widehat{D}}(v, d')$.
  6. If $I = (\mathsf{read}, v, \bot)$ then set $d_{\mathsf{ORAM}} := \textsc{dec}^{\widehat{D}}(v)$.

Detailed Description of Construction Let $\mathsf{ORAM} = (\mathsf{oCompMem}, \mathsf{oCompNext})$ be an ORAM compiler and let $\mathsf{NMCode} = (\textsc{enc}, \textsc{dec}, \textsc{update})$ be a locally decodable and updatable code. We view $\textsc{dec}$ and $\textsc{update}$ as RAM machines and denote by $\Pi_{\textsc{dec}}, \Pi_{\textsc{update}}$ the corresponding next message functions. We present the following tamper and leakage resilient RAM compiler $\mathsf{TLR\text{-}RAM} = (\mathsf{CompMem}, \mathsf{CompNext})$. Here, the parameters $r = r(k), u = u(k), \rho = \rho(k)$ are polynomials in the security parameter k that are implementation-dependent. The complier $\mathsf{TLR\text{-}RAM}$ takes as input an initial database D and a next instruction function Π and does the following:

$\mathsf{CompMem}$: On input security parameter k and initial database D, $\mathsf{CompMem}$ does the following:

- Run $\mathsf{oCompMem}$ to compute $\widetilde{D} \leftarrow \mathsf{oCompMem}(D)$,
  and to initialize $\mathsf{state}_{\mathsf{ORAM}} := (\mathsf{start}, *)$, and $d_{\mathsf{ORAM}} := 0^r$.
- Output $\widehat{D} \leftarrow \textsc{enc}(\widetilde{D})$.
- Initialize $\mathsf{state} \overset{\text{def}}{=} \mathsf{state}_{\mathsf{ORAM}}||\mathsf{state}_{\mathsf{code}}||\mathsf{mode} := (\mathsf{start}, *)||(\mathsf{start}, *)||\bot$
  and $d \overset{\text{def}}{=} d_{\mathsf{code}}||d_{\mathsf{ORAM}} := 0^r||0^r$.

$\mathsf{CompNext}$: On input next instruction function Π, let $\widetilde{\Pi} = \mathsf{oCompNext}(\Pi)$ be the next instruction function of the ORAM compiled machine. $\mathsf{CompNext}(\Pi)$ is the next instruction function of the TLR-RAM compiled machine. It takes as input (state, d) and does the following:

- Parse $\mathsf{state} = \mathsf{state}_{\mathsf{ORAM}}||\mathsf{state}_{\mathsf{code}}||\mathsf{mode}$. Here $\mathsf{mode} \in \{\mathtt{UP}, \mathtt{DEC}, \perp\}$
- If $d_{\mathsf{ORAM}} = \perp$ then abort.
- If $\mathsf{state}_{\mathsf{code}} = (\mathsf{start}, *)$: Compute $(I_{\mathsf{ORAM}}, \mathsf{state}'_{\mathsf{ORAM}}) := \widetilde{\Pi}(\mathsf{state}_{\mathsf{ORAM}}, d_{\mathsf{ORAM}})$.
  1. If I_{ORAM} is of the form $I_{\mathsf{ORAM}} = (\mathsf{wait})$ then set $I := (\mathsf{wait})$.
     Set $\mathsf{state} := \mathsf{state}'_{\mathsf{ORAM}}||\mathsf{state}_{\mathsf{code}}||\mathsf{mode}$.
     Ouput (I, state).
  2. If I_{ORAM} is of the form (stop, z) then set $I := (\mathsf{stop}, z)$.
     Set $\mathsf{state} := \mathsf{state}'_{\mathsf{ORAM}}||\mathsf{state}_{\mathsf{code}}||\mathsf{mode}$.
     Ouput (I, state).
  3. If I_{ORAM} is of the form (write, v, d') then set $\mathsf{state}_{\mathsf{code}} := (\mathsf{start}, v, d')$.
     Set $I := (\mathsf{read}, 0, \perp)$ where $(\mathsf{read}, 0, \perp)$ denotes a dummy read.
     Set $\mathsf{state} := \mathsf{state}_{\mathsf{ORAM}}||\mathsf{state}'_{\mathsf{code}}||\mathtt{UP}$.
     Output (I, state).
  4. If I_{ORAM} is of the form $(\mathsf{read}, v, \perp)$ then set $\mathsf{state}_{\mathsf{code}} := (\mathsf{start}, v)$.
     Set $I := (\mathsf{read}, 0, \perp)$ where $(\mathsf{read}, 0, \perp)$ denotes a dummy read.
     Set $\mathsf{state} := \mathsf{state}_{\mathsf{ORAM}}||\mathsf{state}'_{\mathsf{code}}||\mathtt{DEC}$.
     Output (I, state).
- Otherwise if $\mathsf{state}_{\mathsf{code}} \neq (\mathsf{start}, *)$:
  If $\mathsf{mode} = \mathtt{UP}$, compute $(I_{\mathsf{code}}, \mathsf{state}'_{\mathsf{code}}) := \Pi_{\text{UPDATE}}(\mathsf{state}_{\mathsf{code}}, d_{\mathsf{code}})$.
  If $\mathsf{mode} = \mathtt{DEC}$, compute $(I_{\mathsf{code}}, \mathsf{state}'_{\mathsf{code}}) := \Pi_{\text{DEC}}(\mathsf{state}_{\mathsf{code}}, d_{\mathsf{code}})$.
  1. If I_{code} is of the form (stop, z) then set $I := (\mathsf{read}, 0, \perp)$, where $(\mathsf{read}, 0, \perp)$ denotes a dummy read.
     Set $d_{\mathsf{ORAM}} := z$, set $\mathsf{state}'_{\mathsf{code}} := (\mathsf{start}, *)$, set $\mathsf{state} := \mathsf{state}'_{\mathsf{ORAM}}||\mathsf{state}'_{\mathsf{code}}||\perp$.
     Output (I, state).
  2. If I_{code} is of the form $(\mathsf{read}, \hat{v}, \perp)$, set $I := I_{\mathsf{code}}$.
     Set $\mathsf{state} := \mathsf{state}'_{\mathsf{ORAM}}||\mathsf{state}'_{\mathsf{code}}||\mathtt{DEC}$.
     Output (I, state).
  3. If I_{code} is of the form $(\mathsf{write}, \hat{v}, \hat{d}')$, set $I := I_{\mathsf{code}}$.
     Set $\mathsf{state} := \mathsf{state}'_{\mathsf{ORAM}}||\mathsf{state}'_{\mathsf{code}}||\mathtt{UP}$.
     Output (I, state).
     Upon execution of I, d_{code} will be set to $\widehat{D}[\hat{v}]$.

We are now ready to present the main theorem of this section:

Theorem 4.6. *Assume* $\mathsf{ORAM} = (\mathsf{oCompMem}, \mathsf{oCompNext})$ *is an ORAM compiler which is access-pattern hiding and assume* $\mathsf{NMCode} = (\text{ENC}, \text{DEC}, \text{UPDATE})$ *is a locally decodable and updatable code which is continual non-malleable against* $\mathcal{F}$ *and leakage resilient against* $\mathcal{G}$*. Then* $\mathsf{TLR\text{-}RAM} = (\mathsf{CompMem}, \mathsf{CompNext})$ *presented above is tamper and leakage simulatable w.r.t. function families* $\mathcal{F}, \mathcal{G}$*.*

4.5 Security Analysis

In this section we prove Theorem 4.6. We begin by defining the simulator $\mathcal{S}$. Let $\mathcal{S}_{\mathsf{code}}$ be the simulator guaranteed by the security of $\mathsf{NMCode} = (\textsc{enc}, \textsc{dec}, \textsc{update})$.

For simplicity of exposition, we assume that for every x, given the runtime t of $\mathsf{Execute}(x)$ with respect to $\langle \Pi, D \rangle$, the runtime of $\mathsf{Execute}(x)$ with respect to $\langle \mathsf{oCompNext}(\Pi), \mathsf{oCompMem}(D) \rangle$ is equal to $p(t)$, $p(\cdot)$ is a fixed polynomial known to the simulator. This is indeed the case for the instantiation of our compiler with known underlying building blocks.

Simulator $\mathcal{S}$:

Setup: On input security parameter k, $\mathcal{S}$ does the following:

- Choose a dummy database D_0, compute $\widetilde{D} \leftarrow \mathsf{oCompMem}(D_0)$. Initialize $\mathsf{state}_{\mathsf{ORAM}} := (\mathsf{start}, *)$, $d_{\mathsf{ORAM}} = 0^r$.
- Instantiate the adversary $\mathcal{A}$ and the NMCode simulator $\mathcal{S}_{\mathsf{code}}$.
- Initialize output variable $\mathsf{out} = \perp$ and counter $c = 0$.

Adversarial query $(g, f, \mathsf{doNext}(x))$**:** If $\mathsf{state}_{\mathsf{ORAM}} = (\mathsf{start}, *)$, set $\mathsf{state}_{\mathsf{ORAM}} = (\mathsf{start}, x)$, submit query $\mathsf{Execute}(x)$ to oracle, and receive (z, t). Set $\mathsf{out} = z$ and $c = t$.

Forward (g, f) to $\mathcal{S}_{\mathsf{code}}$. Upon receiving $\mathcal{S}_{\mathsf{code}}$'s output, $(\ell, \mathcal{I}, \vec{w})$, forward ℓ to $\mathcal{A}$.

Case: $\mathcal{I} \neq [n]$**.** Execute a $\mathsf{doNext}(x)$ instruction w.r.t. $\langle \mathsf{oCompNext}(\Pi), \widetilde{D} \rangle$. Let $I_1, \ldots I_{\widetilde{\ell}}$ be the sequence of instructions executed by $\mathsf{doNext}(x)$. Recall that the first $\widetilde{\ell}_1$ instructions are reads, the next $\widetilde{\ell}_2$ instructions are writes, $\widetilde{\ell}_1 + \widetilde{\ell}_2 + 1 = \widetilde{\ell}$ and that $\widetilde{\ell}, \widetilde{\ell}_1, \widetilde{\ell}_2$ are public.

Let $\vec{v} = v_1, \ldots, v_{\widetilde{\ell}-1}$ be the vector of read/write locations corresponding to $I_1, \ldots I_{\widetilde{\ell}}$.

For $1 \leq i \leq \widetilde{\ell}_1$, do the following:

- If $d_{\mathsf{ORAM}} = \perp$ then abort.
- Output $S^{\textsc{dec}}_{v_i}$ to $\mathcal{A}$, where $S^{\textsc{dec}}_{v_i}$ be the ordered set of memory access locations corresponding to $\textsc{dec}(v_i)$. If $v_i \in \mathcal{I}$, set $d_{\mathsf{ORAM}} = \perp$.

For $\widetilde{\ell}_1 + 1 \leq i \leq \widetilde{\ell}_1 + \widetilde{\ell}_2$, $\mathcal{S}$ does the following:

- If $d_{\mathsf{ORAM}} = \perp$ then abort.
- Output $S^{\textsc{update}}_{v_i}$ to $\mathcal{A}$, where $S^{\textsc{update}}_{v_i}$ be the ordered set of memory access locations corresponding to $\textsc{update}(v_i)$. Play the part of the updater interacting with $\mathcal{S}_{\mathsf{code}}$ and submit index v to $\mathcal{S}_{\mathsf{code}}$.

Set $c := c - 1 - \sigma \cdot (\widetilde{\ell}_1 + \widetilde{\ell}_2)$, where σ is the number of instructions in a $\textsc{dec}, \textsc{update}$. If $c = 0$, output out to $\mathcal{A}$ and set $\mathsf{state}_{\mathsf{ORAM}} = (\mathsf{start}, *)$.

Case: $\mathcal{I} = [n]$**.** Do the following until termination:

1. If $d_{\mathsf{ORAM}} = \perp$ then abort.
2. Compute $(I, \mathsf{state}'_{\mathsf{ORAM}}) \leftarrow \mathsf{oCompNext}(\Pi)(\mathsf{state}_{\mathsf{ORAM}}, d_{\mathsf{ORAM}})$. Set $\mathsf{state}_{\mathsf{ORAM}} := \mathsf{state}'_{\mathsf{ORAM}}$.
3. If $I = (\mathsf{wait})$ then set $\mathsf{state}_{\mathsf{ORAM}} := 0^\rho$, $d_{\mathsf{ORAM}} := 0^r$ and terminate.
4. If $I = (\mathsf{stop}, z)$ then set $\mathsf{state}_{\mathsf{ORAM}} = (\mathsf{start}, *)$, $d := 0^r$, output z to $\mathcal{A}$ and terminate.

5. If $I = (\mathsf{read}, v, \perp)$ then set $d_{\mathsf{ORAM}} = \vec{w}_v$. Output S_v^{DEC} to $\mathcal{A}$.
6. If $I = (\mathsf{write}, v, d')$ then do the following: Output S_v^{UPDATE} to $\mathcal{A}$. Play the part of the updater interacting with $\mathcal{S}_{\mathsf{code}}$ and submit index v to $\mathcal{S}_{\mathsf{code}}$.

Lemma 4.7. *Assume* $\mathsf{ORAM} = (\mathsf{oCompMem}, \mathsf{oCompNext})$ *and* $\mathsf{NMCode} = (\text{ENC}, \text{DEC}, \text{UPDATE})$ *are as in Theorem 4.6. Let* Π *be the universal RAM next instruction function. For any ppt adversary* $\mathcal{A}$*, and any initial database* $D \in \{0,1\}^{\text{poly}(k)}$ *we have*

$$\mathbf{TamperExec}(\mathcal{A}, \mathcal{F}, \mathcal{G}, \langle \mathsf{CompNext}(\Pi), \mathsf{CompMem}(D) \rangle) \approx \mathbf{IdealExec}(\mathcal{S}, \langle \Pi, D \rangle)$$

To prove Lemma 4.7 we consider the sequence of hybrids $H_0, H_1, H_{1.5}, H_2$, defined below. We denote by $\mathsf{out}^k_{\mathcal{A},H_i}$, the output distribution of the adversary $\mathcal{A}$ on input security parameter k in Hybrid H_i, for $i \in \{0, 1, 1.5, 2\}$.

Hybrid H_0: This is the simulated experiment $\mathbf{IdealExec}(\mathcal{S}, \langle \Pi, D \rangle)$.

Hybrid H_1: This hybrid is the same as Hybrid H_0 except for the following change is made to the simulator's algorithm: In the Setup stage, the real database D is used to compute $\widetilde{D} \leftarrow \mathsf{oCompMem}(D)$ (instead of $\widetilde{D} \leftarrow \mathsf{oCompMem}(D_0)$).

Claim 4.8.

$$\{\mathsf{out}^k_{\mathcal{A},H_0}\}_{k \in \mathbb{N}} \overset{c}{\approx} \{\mathsf{out}^k_{\mathcal{A},H_1}\}_{k \in \mathbb{N}}.$$

This follows from the security of the ORAM scheme $\mathsf{ORAM} = (\mathsf{oCompMem}, \mathsf{oCompNext})$. Details follow.

Proof. The only difference between the two Hybrids is that in Hybrid H_0 when $\mathsf{doNext}(x)$ is executed, the vector $\vec{v} = v_1, \ldots, v_{\widetilde{\ell}-1}$ is computed using the result of a $\mathsf{doNext}(x)$ instruction w.r.t. $\langle \mathsf{oCompNext}(\Pi), \widetilde{D} \rangle$, where $\widetilde{D} \leftarrow \mathsf{oCompMem}(D_0)$ (and D_0 is the dummy database). On the other hand, in Hybrid H_1, the vector $\vec{v} = v_1, \ldots, v_{\widetilde{\ell}-1}$ is computed using the result of a $\mathsf{doNext}(x)$ instruction w.r.t. $\langle \mathsf{oCompNext}(\Pi), \widetilde{D} \rangle$, where $\widetilde{D} \leftarrow \mathsf{oCompMem}(D)$ (and D is the real initial database). Thus, a distinguisher for Hybrids H_0 and H_1 immediately yields a distinguisher breaking the access pattern hiding property of $\mathsf{ORAM} = (\mathsf{oCompMem}, \mathsf{oCompNext})$. □

Hybrid $H_{1.5}$: We consider the following modification of the Hybrid H_1 experiment:

Upon a $\mathsf{doNext}(x)$ query submitted by the adversary $\mathcal{A}$. If $\mathcal{I} \neq [n]$, execute the following code: (otherwise, the experiment remains unchanged):

Do the following until termination:

1. If $d_{\mathsf{ORAM}} = \perp$ then abort.
2. Compute $(I, \mathsf{state}'_{\mathsf{ORAM}}) \leftarrow \mathsf{oCompNext}(\Pi)(\mathsf{state}_{\mathsf{ORAM}}, d_{\mathsf{ORAM}})$. Set $\mathsf{state}_{\mathsf{ORAM}} := \mathsf{state}'_{\mathsf{ORAM}}$.
3. If $I = (\mathsf{wait})$ then set $\mathsf{state}_{\mathsf{ORAM}} := 0^\rho$, $d_{\mathsf{ORAM}} := 0^r$ and terminate.
4. If $I = (\mathsf{stop}, z)$ then set $\mathsf{state}_{\mathsf{ORAM}} := (\mathsf{start}, *)$, $d := 0^r$ and terminate with output z.
5. If $I = (\mathsf{read}, v, \perp)$ then if $v \notin \mathcal{I}$, set $d_{\mathsf{ORAM}} = \widetilde{D}[v]$. Otherwise, set $d_{\mathsf{ORAM}} = \perp$. Let S_v^{DEC} be the ordered set of memory access locations corresponding to $\text{DEC}(v)$. Output S_v^{DEC} to $\mathcal{A}$.

6. If $I = (\mathsf{write}, v, d')$ then do the following: $\mathcal{S}$ plays the part of the updater interacting with $\mathcal{S}_{\mathsf{code}}$ and submits index v to $\mathcal{S}_{\mathsf{code}}$. Let S_v^{UPDATE} be the ordered set of memory access locations corresponding to $\text{UPDATE}(v)$. Output S_v^{UPDATE} to $\mathcal{A}$.

Claim 4.9.

$$\{\mathsf{out}^k_{\mathcal{A},H_1}\}_{k\in\mathbb{N}} \equiv \{\mathsf{out}^k_{\mathcal{A},H_{1.5}}\}_{k\in\mathbb{N}}.$$

Proof. Intuitively, the difference between Hybrid H_1 and $H_{1.5}$ is that in H_1 in each doNext query, the memory locations $\vec{v} = v_1, \ldots, v_{\widetilde{\ell}-1}$ are pre-computed, whereas in $H_{1.5}$, the memory locations $v_1, \ldots, v_{\widetilde{\ell}-1}$ are computed on the fly. In particular, in H_1, the addresses $\vec{v}$ are computed assuming that each instruction of the form $(\mathsf{read}, v_i, \perp)$ sets d_{ORAM} to the correct value $d_{\mathsf{ORAM}} = \widetilde{D}[v_i]$. On the other hand, in $H_{1.5}$, d_{ORAM} may *not* be set to $\widetilde{D}[v_i]$. However, since we are in the case where $\mathcal{I} \neq [n]$, the only way this can happen is if $v_i \in \mathcal{I}$, in which case d_{ORAM} is set to $\perp$. But now, if $v_i \in \mathcal{I}$, then d_{ORAM} is set to $\perp$ in *both* H_1 and $H_{1.5}$ when the corresponding instruction $(\mathsf{read}, v_i, \perp)$ is simulated. Moreover, once d_{ORAM} is set to $\perp$ then the execution immediately aborts in both H_1 and $H_{1.5}$. Thus, the view of the adversary is identical in H_1 and $H_{1.5}$. □

Hybrid H_2: This is the real experiment $\mathbf{TamperExec}(\mathcal{A}, F, G, \langle \mathsf{CompNext}(\Pi), \mathsf{CompMem}(D)\rangle)$.

Claim 4.10.

$$\{\mathsf{out}^k_{\mathcal{A},H_1}\}_{k\in\mathbb{N}} \stackrel{c}{\approx} \{\mathsf{out}^k_{\mathcal{A},H_2}\}_{k\in\mathbb{N}}.$$

This follows from the security of the locally decodable and updatable code $\mathsf{NMCode} = (\text{ENC}, \text{DEC}, \text{UPDATE})$. Details follow.

Proof. We claim that Hybrid $H_{1.5}$ can be perfectly simulated given the output of $\mathbf{Ideal}_{\mathcal{S},\mathcal{U},M}$, while Hybrid H_2 can be perfectly simulated given the output of $\mathbf{TamperLeak}_{\mathcal{A}',\mathcal{U},M}$, where $\mathcal{A}' = \mathcal{A}$ and $\mathcal{S} = \mathcal{S}$ and $\mathcal{U}$ is the following updater:

The Updater $\mathcal{U}$:

- $\mathcal{U}$ keeps persistent state $\mathsf{state}_{\mathsf{ORAM}}$ which is initialized to $(\mathsf{start}, *)$ and d_{ORAM} which is initialized to 0^r.
- On input $\widetilde{D}$, $\mathcal{U}$ does the following:
- If $d_{\mathsf{ORAM}} = \perp$, then $\mathcal{U}$ aborts.
- Otherwise, $\mathcal{U}$ computes $(I, \mathsf{state}'_{\mathsf{ORAM}}) := \mathsf{oCompNext}(\Pi)(\mathsf{state}_{\mathsf{ORAM}}, d_{\mathsf{ORAM}})$ and sets $\mathsf{state}_{\mathsf{ORAM}} := \mathsf{state}'_{\mathsf{ORAM}}$.
- If I is of the form $(\mathsf{read}, v, \perp)$, then $\mathcal{U}$ sets $d_{\mathsf{ORAM}} = \widetilde{D}[v]$ and outputs $\perp$.
- If I is of the form (write, v, d), then $\mathcal{U}$ outputs (v, d).
- Otherwise, $\mathcal{U}$ outputs $\perp$.

Thus, indistinguishability of hybrids $H_{1.5}$ and H_2 reduces to indistinguishability of $\mathbf{Ideal}_{\mathcal{S},\mathcal{U},M}$ and $\mathbf{TamperLeak}_{\mathcal{A}',\mathcal{U},M}$. This concludes the proof of Claim 4.10. □

Acknowledgement. We thank Yevgeniy Dodis for helpful discussions.

References

[1] Divesh Aggarwal, Yevgeniy Dodis, and Shachar Lovett. Non-malleable codes from additive combinatorics. In *STOC*, 2014. http://eprint.iacr.org/2013/201. 1, 5, 16

[2] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 29–45. Springer, August 2002. 1

[3] Shashank Agrawal, Divya Gupta, Hemanta K. Maji, Omkant Pandey, and Manoj Prabhakaran. Explicit non-malleable codes resistant to permutations. In *Cryptology ePrint Archive, Report 2014/316*, 2014. 1, 5, 16

[4] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545. Springer, December 2000. 10

[5] Mihir Bellare and Phillip Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 317–330. Springer, December 2000. 10

[6] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 513–525. Springer, August 1997. 1

[7] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, 2001. 1

[8] Nishanth Chandran, Bhavana Kanukurthi, and Rafail Ostrovsky. Locally updatable and locally decodable codes. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 489–514. Springer, February 2014. 1

[9] Mahdi Cheraghchi and Venkatesan Guruswami. Capacity of non-malleable codes. In Moni Naor, editor, *ITCS 2014*, pages 155–168. ACM, January 2014. 1, 5, 16

[10] Mahdi Cheraghchi and Venkatesan Guruswami. Non-malleable coding against bit-wise and split-state tampering. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 440–464. Springer, February 2014. 1, 5, 16

[11] Seung Geol Choi, Aggelos Kiayias, and Tal Malkin. BiTR: Built-in tamper resilience. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 740–758. Springer, December 2011. 1, 5, 16

[12] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998. 1

[13] Sandro Coretti, Ueli Maurer, Bjorn Tackmann, and Daniele Venturi. From single-bit to multi-bit public-key encryption via non-malleable codes. In *Cryptology ePrint Archive, Report 2014/324*, 2014. 1

[14] Dana Dachman-Soled and Yael Tauman Kalai. Securing circuits against constant-rate tampering. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 533–551. Springer, August 2012. 1, 5

[15] Dana Dachman-Soled and Yael Tauman Kalai. Securing circuits and protocols against 1/poly(k) tampering rate. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 540–565. Springer, February 2014. 1, 5

[16] Ivan Damgård, Sebastian Faust, Pratyay Mukherjee, and Daniele Venturi. Bounded tamper resilience: How to go beyond the algebraic barrier. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 140–160. Springer, December 2013. 1

[17] Yevgeniy Dodis and Krzysztof Pietrzak. Leakage-resilient pseudorandom functions and side-channel attacks on Feistel networks. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 21–40. Springer, August 2010. 1, 5

[18] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, May 2014. 1, 5

[19] Stefan Dziembowski and Sebastian Faust. Leakage-resilient cryptography from the inner-product extractor. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 702–721. Springer, December 2011. 1, 5

[20] Stefan Dziembowski and Sebastian Faust. Leakage-resilient circuits without computational assumptions. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 230–247. Springer, March 2012. 1, 5

[21] Stefan Dziembowski, Tomasz Kazana, and Maciej Obremski. Non-malleable codes from two-source extractors. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 239–257. Springer, August 2013. 1, 5, 16

[22] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *49th FOCS*, pages 293–302. IEEE Computer Society Press, October 2008. 1, 5

[23] Stefan Dziembowski, Krzysztof Pietrzak, and Daniel Wichs. Non-malleable codes. In Andrew Chi-Chih Yao, editor, *ICS 2010*, pages 434–452. Tsinghua University Press, January 2010. 1, 2, 5, 6, 16, 29

[24] Sebastian Faust, Pratyay Mukherjee, Jesper Buus Nielsen, and Daniele Venturi. Continuous non-malleable codes. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 465–488. Springer, February 2014. 1, 5, 8, 16

[25] Sebastian Faust, Pratyay Mukherjee, Jesper Buus Nielsen, and Daniele Venturi. A tamper and leakage resilient random access machine. In *Cryptology ePrint Archive, Report 2014/338*, 2014. 5

[26] Sebastian Faust, Pratyay Mukherjee, Daniele Venturi, and Daniel Wichs. Efficient non-malleable codes and key-derivation for poly-size tampering circuits. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 111–128. Springer, May 2014. 1, 5, 16

[27] Sebastian Faust, Krzysztof Pietrzak, and Daniele Venturi. Tamper-proof circuits: How to trade leakage for tamper-resilience. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part I*, volume 6755 of *LNCS*, pages 391–402. Springer, July 2011. 1, 5

[28] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting circuits from leakage: the computationally-bounded and noisy cases. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 135–156. Springer, May 2010. 1, 5

[29] Rosario Gennaro, Anna Lysyanskaya, Tal Malkin, Silvio Micali, and Tal Rabin. Algorithmic tamper-proof (ATP) security: Theoretical foundations for security against hardware tampering. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 258–277. Springer, February 2004. 1

[30] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996. 4

[31] Shafi Goldwasser and Guy N. Rothblum. Securing computation against continuous leakage. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 59–79. Springer, August 2010. 1, 5

[32] Shafi Goldwasser and Guy N. Rothblum. How to compute in the presence of leakage. In *53rd FOCS*, pages 31–40. IEEE Computer Society Press, October 2012. 1, 5

[33] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, pages 45–60, 2008. 1

[34] Yuval Ishai and Eyal Kushilevitz. On the hardness of information-theoretic multiparty computation. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 439–455. Springer, May 2004. 1

[35] Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and David Wagner. Private circuits II: Keeping secrets in tamperable circuits. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 308–327. Springer, May / June 2006. 1, 5

[36] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, August 2003. 1, 5

[37] Ali Juma and Yevgeniy Vahlis. Protecting cryptographic keys against continual leakage. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 41–58. Springer, August 2010. 1, 5

[38] Jonathan Katz and Luca Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *32nd ACM STOC*, pages 80–86. ACM Press, May 2000. 1

[39] Jonathan Katz and Moti Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In Bruce Schneier, editor, *FSE 2000*, volume 1978 of *LNCS*, pages 284–299. Springer, April 2000. 10

[40] Aggelos Kiayias and Yiannis Tselekounis. Tamper resilient circuits: The adversary at the gates. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 161–180. Springer, December 2013. 1, 5

[41] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, August 1996. 1

[42] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, August 1999. 1

[43] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, pages 168–177, 2000. 5

[44] Feng-Hao Liu and Anna Lysyanskaya. Tamper and leakage resilience in the split-state model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 517–532. Springer, August 2012. 1, 3, 5, 6, 8, 16

[45] Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 278–296. Springer, February 2004. 1, 5

[46] Krzysztof Pietrzak. A leakage-resilient mode of operation. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 462–482. Springer, April 2009. 1, 5

[47] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 09*, pages 199–212. ACM Press, November 2009. 1

[48] Guy N. Rothblum. How to compute under AC^0 leakage without secure hardware. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 552–569. Springer, August 2012. 1, 5

[49] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing, ICS 2003*, pages 160–171, 2003. 5

[50] Amit Vasudevan, Jonathan M. McCune, James Newsome, Adrian Perrig, and Leendert van Doorn. CARMA: a hardware tamper-resistant isolated execution environment on commodity x86 platforms. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 48–49. ACM Press, May 2012. 5

[51] Sergey Yekhanin. Locally decodable codes. *Foundations and Trends in Theoretical Computer Science*, 6(3):139–255, 2012. 1

A Strong Non-malleability

Here we first recall the Strong Non-malleability notion originally defined by Dziembowski et al. [23]. Then we define Strong Non-malleability against one-time, and continual attacks respectively. We remark that our constructions in Section 3 can achieve the stronger notion of non-malleability if the underlying non-malleable code is the stronger one. We defer the rigorous analysis to the later version of this paper.

Definition A.1 (Strong Non-malleability [23]). *Let k be the security parameter, $\mathcal{F}$ be some family of functions. For each function $f \in \mathcal{F}$, and $m \in \Sigma$, define the tampering experiment*

$$\mathbf{StrongNM}_m^f \stackrel{\text{def}}{=} \left\{ \begin{array}{c} c \leftarrow \text{ENC}(m), \tilde{c} := f(c), \tilde{m} := \text{DEC}(\tilde{c}) \\ Output: \ \mathsf{same}^* \ if \ \tilde{c} = c, \ and \ \tilde{m} \ otherwise. \end{array} \right\}$$

The randomness of this experiment comes from the randomness of the encoding algorithm. We say that a coding scheme (ENC, DEC) *is strong non-malleable with respect to the function family $\mathcal{F}$ if for any $m, m' \in \Sigma$ and for each $f \in \mathcal{F}$, we have:*

$$\{\mathbf{StrongNM}_m^f\}_{k \in \mathbb{N}} \approx \{\mathbf{StrongNM}_{m'}^f\}_{k \in \mathbb{N}}$$

where $\approx$ can refer to statistical or computational indistinguishability.

One time security Strong Non-malleability against one-time physical attacks is defined as follows.

Definition A.2 (Strong Non-malleability of Locally Decodable Codes). *Let k be the security parameter, $\mathcal{F}$ be some family of functions. For each function $f \in \mathcal{F}$, and $M = (m_1, m_2, \ldots, m_n) \in \Sigma^n$, define the tampering experiment*

$$\mathbf{StrongNM}_M^f \stackrel{\text{def}}{=} \left\{ \begin{array}{l} C \leftarrow \text{ENC}(M), \tilde{C} = f(C), \tilde{m}_i = \text{DEC}^{\tilde{C}}(i) \ for \ i \in [n]. \\ If \ \exists \ i \ such \ that \ \tilde{m}_i \neq \perp \ \& \ \tilde{C} \ and \ C \ are \ not \ identical \ for \ all \ queries \ by \ \text{DEC}(i), \\ \qquad then \ output: \ (\tilde{m}_1, \tilde{m}_2, \ldots, \tilde{m}_n). \\ Else, \ set \ m'_i = \mathsf{same}^* \ if \ C \ and \ \tilde{C} \ are \ identical \ for \ all \ queries \ of \ \text{DEC}(i); \\ \qquad otherwise \ m'_i = \perp. \ Then \ output: \ (m'_1, m'_2, \ldots, m'_n). \end{array} \right\}$$

The randomness of this experiment comes from the randomness of the encoding and decoding algorithms.

We say that a locally decodable coding scheme (ENC, DEC, UPDATE) *is strong non-malleable against the function class $\mathcal{F}$ if for any $M, M' \in \Sigma^n$ and for any $f \in \mathcal{F}$, we have:*

$$\{\mathbf{StrongNM}_M^f\}_{k \in \mathbb{N}} \approx \{\mathbf{StrongNM}_{M'}^f\}_{k \in \mathbb{N}}$$

where $\approx$ can refer to statistical or computational indistinguishability.

Continual security Strong Non-malleability against continual physical attacks is defined as follows.

Definition A.3 (Strong Continual Tampering and Leakage Experiment). *Let k be the security parameter, $\mathcal{F}, \mathcal{G}$ be some families of functions. Let* (ENC, DEC, UPDATE) *be an $(n, \hat{n}, p, q)$-locally decodable and updatable coding scheme with respect to $\Sigma, \hat{\Sigma}$. Let $\mathcal{U}$ be an updater that takes* *input a message M and outputs an index $i \in [n]$ and $m \in \Sigma$. Then for any blocks of messages $M = (m_1, m_2, \ldots, m_n) \in \Sigma^n$, and any (non-uniform) adversary $\mathcal{A}$, any updater $\mathcal{U}$, define the following experiment* $\mathbf{StrongTL}_{\mathcal{A},\mathcal{U},M}$:

- *The challenger first computes an initial encoding $C^{(1)} \leftarrow \textsc{enc}(M)$.*
- *Then the following procedure repeats, at each round j, let $C^{(j)}$ be the current codeword and $M^{(j)}$ be the underlying message:*
  - *$\mathcal{A}$ sends either a tampering function $f \in \mathcal{F}$ and/or a leakage function $g \in \mathcal{G}$ to the challenger.*
  - *The challenger replaces the codeword with $f(C^{(j)})$, or sends back a leakage $\ell^{(j)} = g(C^{(j)})$.*
  - *Then we define $\vec{m}^{(j)}$ for the following two conditions:*
    - *If there exists i such that $\textsc{dec}^{f(C^{(j)})}(i) \neq \perp$ and $C^{(j)}$ and $f(C^{(j)})$ are not identical for all queries from $\textsc{dec}(i)$, then set $\vec{m}^{(j)} = (\textsc{dec}^{f(C^{(j)})}(1), \ldots, \textsc{dec}^{f(C^{(j)})}(n))$.*
    - *Else, for $i \in [n]$, let $m'_i = \mathsf{same}^*$ if $f(C^{(j)})$ and $C^{(j)}$ are identical for all queries of $\textsc{dec}(i)$, otherwise $m'_i = \perp$. Then set $\vec{m}^{(j)} = (m'_1, m'_2, \ldots, m'_n)$.*
  - *Then the updater sends $(i^{(j)}, m) \leftarrow \mathcal{U}(M^{(j)})$ to the challenger, and the challenger runs $\textsc{update}^{f(C^{(j)})}(i^{(j)}, m)$ and sends the index $i^{(j)}$ to $\mathcal{A}$.*
  - *$\mathcal{A}$ may terminate the procedure at any point.*
- *Let t be the total number of rounds above. At the end, the experiment outputs*

$$\left(\ell^{(1)}, \ell^{(2)}, \ldots, \ell^{(t)}, \vec{m}^{(1)}, \ldots, \vec{m}^{(t)}, i^{(1)}, \ldots, i^{(t)}\right).$$

Definition A.4 (Strong Non-malleability and Leakage Resilience against Continual Attacks)**.** *An $(n, \hat{n}, p, q)$-locally decodable and updatable coding scheme with respect to $\Sigma, \hat{\Sigma}$ is strong continual non-malleable against $\mathcal{F}$ and leakage resilient against $\mathcal{G}$ if for all* PPT *(non-uniform) adversaries $\mathcal{A}$, any* PPT *updater $\mathcal{U}$, any messages $M, M' \in \Sigma^n$, the experiments* $\mathbf{StrongTL}_{\mathcal{A},\mathcal{U},M}$ *and* $\mathbf{StrongTL}_{\mathcal{A},\mathcal{U},M'}$ *are (computationally) indistinguishable.*