# A Tamper and Leakage Resilient Random Access Machine

Sebastian Faust[1], Pratyay Mukherjee[2], Jesper Buus Nielsen[2], and Daniele Venturi[3]

[1]*EPFL Switzerland*
[2]*Aarhus University*
[3]*Sapienza University of Rome*

May 14, 2014

**Abstract**

We present a "universal" Random Access Machine (RAM in short) for tamper and leakage resilient computation. The RAM has one CPU that accesses three storages (called *disks* in the following), two of them are secret, while the other one is public. The CPU has *constant size* for each fixed value of security parameter $k$. We construct a compiler for this architecture which transforms any keyed primitive into a RAM program where the key is encoded and stored on the two secret disks and the instructions for evaluating the functionality are stored on the public disk.

The compiled program tolerates arbitrary *independent* tampering of the disks. That is, the adversary can tamper with the intermediate values produced by the CPU, and the *program code* of the compiled primitive on the public disk. In addition, it tolerates bounded independent leakage from the disks and *continuous leakage* from the communication channels between the disks and the CPU.

Although it is required that the circuit of the CPU is tamper and leakage proof, its design is independent of the actual primitive being computed and its internal storage is non-persistent, i.e., all secret registers are reset between invocations. Hence, our result can be interpreted as reducing the problem of shielding arbitrary complex computations to protecting a single, simple and "universal" component. As a main ingredient of our construction we use continuous non-malleable codes that satisfy certain additional properties.

## Contents

# 1 Introduction

Can cryptographic schemes achieve their security goals when run on non-trusted machines? This fascinating question has recently resulted in a large body of work that weakens the traditional assumption of fully trusted computation and gives the adversary partial control over the implementation. Such partial control can either be *passive* where the adversary obtains information about the internal computation, or *active* where the adversary is allowed to change the secret state and/or the computation of the scheme. While the question of whether cryptography is possible in a non-trusted environment is interesting by itself, it also has important practical applications for, e.g., protecting cryptographic implementations against so-called side-channel [27, 28], fault [3, 7, 35, 33] or virus attacks [32, 21].

One general solution to the above question is given by the appealing notion of leakage and tamper resilient compilers introduced in the pioneering works of Ishai, Prabhakaran, Sahai and Wagner [24, 23]. A compiler takes as input a description of some arbitrary cryptographic functionality $\mathcal{G}_{\mathsf{K}}$ and outputs a transformed functionality $\mathcal{G}'_{\mathsf{K}'}$ which has the same input/output behaviour as $\mathcal{G}_{\mathsf{K}}$ but additionally remains secure in a partially trusted environment. For instance, $\mathcal{G}'_{\mathsf{K}'}$ may be secure when the adversary is able to obtain a bounded amount of leakage from the execution of $\mathcal{G}'_{\mathsf{K}'}$, or when he can change the secret state $\mathsf{K}'$ in some adversarial way. Formally, security is typically modelled by a simulation-based notion. That is, whatever the adversary can learn by interacting with $\mathcal{G}'_{\mathsf{K}'}$ in the non-trusted environment, he can also achieve by interacting with the original $\mathcal{G}_{\mathsf{K}}$ when implemented on a fully trusted device.

**Tamper resilient compilers.** Two different lines of work investigate methods for tamper resilient compilers. The first approach designs so-called tamper resilient circuits [23, 19, 11, 26, 12]. That is, given a circuit $C[\mathsf{K}]$ that, e.g., computes the AES with key $\mathsf{K}$, the compiler outputs[1] a transformed circuit $C'[\mathsf{K}']$ that achieves simulation-based security even if the adversary can tamper with up to a constant fraction of the wires independently. While these works allow the adversary to tamper with the entire circuitry, they typically make very strong assumptions on the type of tampering. In particular, it is assumed that each bit of the computation is tampered with independently (so-called set/reset and toggle attacks).

The second approach is based on the beautiful notion of non-malleable codes [16]. Informally, a code is non-malleable w.r.t. a set of tampering functions if the message contained in a codeword modified via a function in the family is either the original message, or a completely "unrelated" value. A compiler based on non-malleable codes stores the secret key in an encoded form and the compiled functionality decodes the state each time the functionality wants to accesses the key. As long as the adversary can only apply tampering functions from the family supported by the code, the non-malleability property guarantees that the (possibly tampered) decoded value is not related to the original key. While non-malleable codes exist for rich families that go far beyond the bit-tampering adversary discussed above (see, e.g., [16, 30, 15, 1, 8, 9, 17, 18, 2, 10]), the existing compilers based on non-malleable codes only protect the secret key against tampering attacks. In particular, the assumption is that the entire circuitry that evaluates the functionality is implemented on a fully trusted environment and cannot be tampered with.

In this work we show how to *significantly* weaken the assumption of tamper-proof computation. Our solution is also based on non-malleable codes and hence can achieve strong protection against

[1] W.l.o.g. we assume that any keyed-functionality $\mathcal{G}_{\mathsf{K}}(\cdot)$ can be realized by a corresponding Boolean circuit $C_{\mathcal{G}}[\mathsf{K}]$.

rich families of tampering functions, but simultaneously significantly reduces the assumption on tamper proof circuitry used by the traditional approach described above. In particular, the tamper-proof circuitry we use (the so-called CPU) is a *small* and *universal* component, whose size and functionality is *independent* of the functionality that we want to protect. Notice that this is in contrast to the traditional approach that requires a specifically tailored tamper-proof hardware for each functionality that we intend to protect. Our solution is hence in spirit with earlier works (e.g., [19]) and reduces the problem of protecting arbitrary complicated computation to shielding a single, simple component. While our construction may work generically for non-malleable codes that satisfy certain properties, in this work we focus on non-malleable codes in the split-state setting.[2] In this well-known (considered in [30, 1, 15, 17, 9] etc.) setting the codeword consists of two parts $c_0, c_1$ and the adversary is allowed to tamper independently with them in an arbitrary way.

**On the difficulty of computing with non-malleable codes.** As non-malleable codes typically do not have any homomorphic property that enables computation, and in fact in many cases a simple homomorphism would contradict the non-malleability property of the code, our tamper proof CPU will carry out the decoding and encoding procedure of the underlying non-malleable code. Additionally, it will execute a single constant size instruction of the functionality. More concretely, consider some instruction $\mathcal{I}$ with inputs $c_\alpha = \text{Encode}(\alpha)$ and $c_\beta = \text{Encode}(\beta)$ (e.g., $\mathcal{I}$ may be the NAND operation and $c_\alpha$ and $c_\beta$ may be the encodings of two bits $\alpha$ and $\beta$), then informally the tamper proof CPU will carry out the following operations: (i) Decode $c_\alpha$ and $c_\beta$ to recover the bits $\alpha$ and $\beta$; (ii) Compute the instruction $\mathcal{I}$ on inputs $\alpha$ and $\beta$ to obtain the result $\gamma$; (iii) Compute $c_\gamma = \text{Encode}(\gamma)$. Notice that instruction $\mathcal{I}$ may be as simple as a single NAND operation, and hence the size of the CPU is *independent* of the size of the transformed functionality.

One may think that given such a powerful tamper-proof component a solution for tamper resilient computation is simple. Unfortunately, the notion of non-malleable codes only guarantees that one cannot change the encoded value to some related value. Nothing, however hinders the adversary to just overwrite an encoding with a valid encoding of some fixed (known) value. Notice that such an attack may not only make it impossible to achieve simulation-based security, but moreover can completely break the scheme.[3] In the split state model where all encodings are stored on two independent disks, the situation is even more severe as now the adversary can start to copy valid encodings from some place of the computation to different portions. For instance, he may attempt to copy the encoding of the secret key directly to the output of the program. Our transformation prevents these and other attacks by tying together all encodings with the secret key and the description of the compiled functionality. Hence, any attempt to change any intermediate encoding will destroy the functionality, including the key.

One important feature of our construction is to allow tampering with the program code. In our model the program consists of code built from several instructions such that each instruction is executed by the the tamper-proof CPU sequentially. Notice that tampering with the program (and hence with the functionality) is allowed as the code is written on the tamperable public disk. Hence, the adversary may attempt to overwrite the code with a malicious program that, e.g., just outputs the secret key. In our construction we prevent this type of attack by again making sure

[2] It would be interesting to analyze if our construction can be extended to work with other (non split-state) non-malleable codes, as it is the case for the compiler of [16].

[3] Consider a contrived program that outputs the secret key if a certain status bit is set to 0, but otherwise behaves normally.

that any change of the code will enforce in tampering with the secret key, which itself is protected by a non-malleable code.

**On the difficulty of adding leakage.** While the CPU is assumed to be fully trusted, i.e., its computation neither can be changed nor may it leak information to an adversary, the data read from and written to the disks may leak. We model this by giving the adversary bounded leakage from the *buses*, which are basically the communication channels carrying data, read from and written to the disks. We emphasize that our leakage model is *continuous* in the sense that the CPU can be executed many times and hence over time the leakage from the read/write process may exceed the size of the disks. It is well known that leakage and tamper resilience do not add up [29]. In fact, it is easy to construct a scheme that is leakage resilient but completely breaks under limited tampering attack. In this context, one possible attack that we need to prevent is a so-called *resetting* attack: an adversary may keep a copy of the entire disks and re-run the program continuously on the same state. This may result in loading over and over again the same data, which is eventually revealed through continuous leakage. As reset attacks seem inherent in a model where both tampering and leakage is possible (the tampering allows resetting and the leakage enables the adversary to eventually learn the entire state), we require that the CPU keeps a small (logarithmic in the number of executions) and public but tamper-proof state—a so-called activation and program counter. Our construction has then to ensure that the public counters are synchronized with the current state kept on the disks.

In the next two subsections we provide a high level description of our model and techniques. For a comparison of our result with other results in the area of tamper resilient cryptography (in particular with tamper resilient circuits), we refer the reader to Section 1.3.

## 1.1 Our Model

We put forward a model of a tamper and leakage resilient Random Access Machine (RAM) architecture, which can implement arbitrary keyed functionalities $\mathcal{G}_{\mathsf{K}}(\cdot)$.

**Split-state RAM schemes.** Our split-state RAM schemes consist of a RAM architecture $\mathbf{R}$ and a RAM compiler $\mathbf{C}$. The RAM $\mathbf{R}$ has two secret disks $\mathcal{SD}_1, \mathcal{SD}_2$, one public disk $\mathcal{PD}$ and a tamper/leakage-proof CPU that is connected with the disks through buses. The RAM compiler $\mathbf{C}$ takes as input the description of a functionality $\mathcal{G}$ and a key $\mathsf{K}$ and outputs initial states for the execution of the program in the RAM architecture. In particular, this consists of the contents for the public and secret disks. The former contains an encoding of the program code, while the secret disks keep the encoding of $\mathsf{K}$.

The program runs in *activations*. An activation denotes the time period of evaluating $\mathcal{G}_{\mathsf{K}}(\cdot)$ on some input $x$, where $x$ is stored on $\mathcal{PD}$. An activation involves several *executions* of the CPU. In each execution, the CPU loads a constant number of encodings from the disks (this might include reading part of the input), executes one computation (including potential decoding and encoding) on the loaded data, and writes the result back to the disks (this might include writing part of the output). We stress that our CPU has no persistent internal (secret) storages, i.e., all secret registers are reset between invocations.

As discussed above, to achieve our strong security guarantee of continuous leakage and split-state tamper resistance, the CPU contains a public untamperable program counter $\mathtt{pc}$, an activation

counter ac and a self-destruct bit B. While the first two are necessary to prevent the reset attack in the presence of continuous leakage, the self-destruct bit is required for continuous tamper resilience (this is similar to earlier works that build compilers based on non-malleable codes [17]). The activation counter ac is incremented after each activation and during each activation the program counter pc specifies at which position of the public disk the CPU shall read the next instruction. The value B is a special self-destruct bit that is initially set to 0, and can once be flipped by the CPU. Whenever B is set to 1, the RAM goes into a special "self-destruct" mode where it is assumed to forever output the all-zero string.

**RAM-simulatability.** To define tamper resilience of a RAM-scheme, we introduce a natural notion of RAM-simulatability for split-state RAM-schemes. At setup, the RAM-compiler is run in order to produce the initial contents of the public and secret disks. As in previous works on tamper and leakage resilient compilers the preprocessing in the setup is assumed to be tamper and leakage proof and is executed once at the initialization of the system. In the online phase, the adversary can specify between executions a tampering function $\mathsf{Tamper}(\cdot)$ that modifies the public disks $\mathcal{PD}$ and the two secret disks $\mathcal{SD}_1, \mathcal{SD}_2$ *independently* (yet arbitrarily).[4] Furthermore, the adversary can ask the RAM to perform the next step in the computation (for the current activation), by running the CPU on the (possibly modified) disks. We call this experiment the *real execution*. We remark that since the adversary can modify arbitrarily the public disk, our model allows arbitrary tampering with the *program code* of the (transformed) functionality.

We compare the real execution to a mental experiment featuring a simulator having only black-box access to the original functionality $\mathcal{G}_{\mathsf{K}}(\cdot)$. We call this an *ideal execution*. A split-state RAM-scheme is RAM-simulatable if for all efficient adversaries there exists an efficient simulator such that for all functionalities $\mathcal{G}$ the output distributions of a real and an ideal execution are computationally close.

**On the trusted CPU assumption.** The CPU is the only part of the computation that is completely trusted. While its inputs and outputs may be subject to leakage and tampering attacks, its computation does not leak and its execution is carried out un-tampered. Our CPU is small and independent of the functionality to protect: it merely reads a constant number of encodings from disks, decodes them, executes some instruction (that can be as simple as a NAND operation) and writes the encoded result back to the disks. Notice that in contrast to earlier work on tamper resilient compilers based on non-malleable codes [16, 30, 17], we allow tampering with intermediate values produced by the program code, and in fact even with the program code itself. Our result hence can be interpreted as a much more granular model of computation than [16, 30, 17]. In summary, we show how to reduce the problem of protecting arbitrary computation against continuous leakage and tampering attacks in the split-state model to shielding a *simple* and *universal* component. We notice that while our work minimizes the trusted hardware assumption made in non-malleable code based compilers, our trusted CPU is significantly more complex than tamper-proof hardware that has been used in works on tamper resilient circuits (cf. Section1.3 for more details on this).

[4]Note that, since $\mathcal{PD}$ is public, it is always possible to tamper jointly with the public disk and either of $\mathcal{SD}_i$.

## 1.2 Our Techniques

The description of our RAM-scheme consists of two modular steps. In the first step, we build a compiler achieving only a weaker form of RAM-simulatability, where the way an adversary can tamper with the RAM is significantly limited. In the second step, we show how to take a scheme secure in this "hybrid world" and secure it against the tampering possible in the split-state model via what we call an emulator. The combination of the two transformations yields our final split-state RAM-scheme. We first describe the emulator first.

**The emulator.** The RAM architecture in the hybrid-world only has one secret disk $\mathcal{SD}^{\mathsf{h}}$ and one public disk $\mathcal{PD}^{\mathsf{h}}$. While the adversary has full read/write access to $\mathcal{PD}^{\mathsf{h}}$, he only has limited tampering access to $\mathcal{SD}^{\mathsf{h}}$: he may copy values within $\mathcal{SD}^{\mathsf{h}}$ itself, and replace parts of $\mathcal{SD}^{\mathsf{h}}$ with some known values. We call the modified execution the *hybrid execution*. We then introduce the notion of a hybrid-to-real emulator, which is essentially an efficient transformation taking as input a program secure in the hybrid world and a program secure in the real world, i.e., on the split-state RAM.

The basic idea of the hybrid emulator is simple. Given a hybrid scheme, each value of the disks is encoded using a suitable non-malleable code. Our construction uses the recent construction of continuous non-malleable codes (CNMC) in the split-state setting [17]. In contrast to traditional non-malleable codes, continuous non-malleability guarantees that the code remains secure under continuous attacks without assuming erasures. Our construction requires also some form of composability of non-malleable codes, where we allow the tampering function to depend on multiple encodings together. We can show by a generic reduction that composability is preserved for any continuous non-malleable split-state code. Finally, the emulator transforms the program code by a straightforward transformation that guarantees that the required encodings are loaded from the correct positions of the two secret disks (instead of the single disk in the hybrid scheme).

We show by a reduction to the composable CNMC that there exists a hybrid simulator, attacking the hybrid scheme and having limited tamper access (only copy and replace), that produces a distribution that is indistinguishable from the execution of the emulated RAM scheme in the real world. For this reduction to work, it is important that the hybrid scheme being emulated has a property called $c$-boundedness. Informally, this notion says that each value on the secret disk is touched at most $c$ times, for a constant $c$. Without this property, the emulator would touch the corresponding codeword an unbounded number of times, and continuous leakage from the buses would leak the entire code. Notice that it is in particular difficult to achieve $c$-bounded schemes in the presence of tampering, as the hybrid adversary may several times move a given value to the next position on the secret disk read by the CPU. The proof of the emulator is quite tedious and requires some careful book-keeping.

**The hybrid compiler.** We construct a hybrid-scheme that is RAM-simulatable and $c$-bounded for a small constant $c$. The main idea is to store the program and all intermediate values on the secret disk $\mathcal{SD}^{\mathsf{h}}$. At setup, a secret label $L$ is sampled uniformly at random and stored in the first position of the secret disk. Then, each value on the disk is "augmented" with the following information: (i) The position $j$ at which the value was meant to be stored; (ii) The secret label $L$; and (iii) The values $(a, p)$ of the activation counter `ac` and the program counter `pc` when the value was written on disk. Intuitively, adding the secret label (which is unknown to the adversary) prevents the adversary from replacing values from different positions of the secret disk with values

that do not have the right label (notice that this label is long enough such that it cannot be guessed by the adversary). This ensures that all the values containing the label are either from the pre-processing or computed and stored by the CPU. Hence, they are in a way "authenticated" by the computation and not introduced by the adversary. On the other hand, the position $j$ prevents the adversary from copying the corresponding value to a location different from $j$, as the CPU will check that $j$ matches the position from which the value was read.

Note that the adversary can still replace a value at location $j$ with an older value that was stored at location $j$ before, essentially with the goal of resetting the scheme to a previous valid state. By checking the values $a$ and $p$ with the current values of the activation and program counters of the CPU, the CPU can detect such resetting attacks and self-destruct if necessary. Our analysis (see Section 5) shows that the probability that an adversary manages to replace some value on the secret disk (putting the correct label) without generating a self-destruct, is exponentially small in the security parameter. The use of the label to prevent moving and resetting values along with the structure of the compiled program makes or hybrid compiler $c$-bounded, as require by the emulator.

### 1.3 Other Related Work

Many recent works have studied the security of specific cryptographic schemes (e.g., public key encryption, signatures or pseudorandom functions) against tampering attacks [5, 4, 25, 36, 6, 13]). While these works often consider a stronger tampering model and make less assumptions about tamper-proof hardware, they do not work for arbitrary functionalities.

**Leakage and tamper-proof circuits.** A large body of work studies the security of Boolean circuits against leakage attacks [24, 20, 14, 22, 34, 31]. While most works on leakage resilient circuit compilers require leakage-proof hardware, the breakthrough work of Goldwasser and Rothblum [22] shows how to completely eliminate leak-proof hardware for leakage in the split-state setting. It is an interesting open question, if one can use the compiler of [22] to implement our CPU and allow leakage also from its execution. We emphasize that most of the work on leakage resilient circuit compilers does not consider tampering attacks (though some of them may be easily extendible in restricted tampering settings [23]).

The concept of tamper resilient circuits has been introduced by Ishai, Prabhakaran, Sahai and Wagner [23] and further studied in [23, 19, 11, 26, 12]. On the upside such compilers require much simpler tamper-proof hardware,[5] but study a much weaker tampering model. Concretely, they assume that an adversary can tamper with individual wires (or constant size gates [26]) independently. That is, the adversary can set the bit carried on a wire to 1, set it to 0 or toggle its value. Moreover, it is assumed that in each execution at least a constant fraction of the wires is not tampered at all.[6] Our model considers a much richer family of tampering attacks. In particular, we allow the adversary to *arbitrarily* tamper with the entire content of the two disks, as long as the tampering is done independently. In fact, our model even allows the adversary to tamper with the functionality by putting the program code on the public disk to which the adversary has complete read/write access. Translating this power to a circuit model would essentially allow the adversary to "re-wire" the circuit.

[5] To the best of our knowledge each of these compilers requires a tamper-proof gate that operates on at least $k$ inputs where $k$ is the security parameter. Asymptotically, this is also the case for our CPU, while clearly from a practical perspective our tamper-proof hardware is significantly more complex.

[6] In [23, 19] it is allowed that faults are persistent so at some point the entire circuitry may be subject to tampering.

Finally we notice that our RAM model can be thought of, in fact, as a generalization of the circuit model where the RAM program can be, e.g., a Boolean circuit and the CPU evaluates NAND gates.

# 2 Preliminaries

## 2.1 Notation

For $n \in \mathbb{N}$, we write $[n] := \{1, \ldots, n\}$. Given a set $\mathcal{S}$, we write $s \leftarrow \mathcal{S}$ to denote that element $s$ is sampled uniformly from $\mathcal{S}$. If $\mathsf{S}$ is an algorithm, $y \leftarrow \mathsf{S}(x)$ denotes an execution of $\mathsf{S}$ with input $x$ and output $y$; if $\mathsf{S}$ is randomized, then $y$ is a random variable. Let $k \in \mathbb{N}$ be a security parameter. We use $negl(k)$ to denote a negligible function on $k$. Given two random variables $X_1$ and $X_2$, we write $X_1 \overset{\text{c}}{\approx} X_2$ to denote that $X_1$ and $X_2$ are computationally indistinguishable meaning that for all PPT algorithms $\mathcal{A}$ we have that $\Pr[\mathcal{A}(X_1) = 1] - \Pr[\mathcal{A}(X_2) = 1] \leq negl(k)$.

## 2.2 Continuous Non-Malleable Codes

In this paper we consider non-malleable codes in the split-state setting and omit to mention it explicitly for the rest of the paper. A split-state encoding scheme $\mathcal{C} = (\text{Init}, \text{Encode}, \text{Decode})$, is a triple of algorithms specified as follows: (1) Init, takes as input the security parameter and outputs public parameters $\Omega \leftarrow \text{Init}(1^k)$; (2) Encode, takes as input a string $x \in \{0,1\}^*$ and the public parameters, and outputs a codeword $c = (c_0, c_1) \leftarrow \text{Encode}(\Omega, x)$ where $c \in \{0,1\}^{2n}$; (3) Decode, takes as input a codeword $c \in \{0,1\}^{2n}$ and the public parameters, and outputs a value $x = \text{Decode}(c)$ where $x \in \{0,1\}^* \cup \{\perp\}$. We require that $\text{Decode}(\Omega, \text{Encode}(\Omega, x)) = x$ for all $\Omega \leftarrow \text{Init}(1^k)$ and for all $x \in \{0,1\}^*$. Moreover, for any two inputs $x_0, x_1$ ($|x_0| = |x_1|$) and any efficient function $\mathsf{T}_0, \mathsf{T}_1$ the probability that the adversary guesses the bit $b$ in the following game is negligible: (i) Sample $b \leftarrow \{0,1\}$ and compute $(c_0, c_1) \leftarrow \text{Encode}(\Omega, x_b)$, and (ii) the adversary obtains $\text{Decode}^*(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1))$, where $\text{Decode}^*$ is as Decode except that it returns a special symbol $\mathsf{same}^\star$ if $(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1)) = (c_0, c_1)$.

The above one-shot game has been extended to the continuous setting in [17], where the adversary may tamper continuously with the encoding. In contrast to the above game, the adversary here obtains access to the following tampering oracle $\mathcal{O}^q_{\mathsf{cnm}}((c_0, c_1), \cdot)$, where $(c_0, c_1)$ is an encoding of either $x_0$ or $x_1$:

$\underline{\mathcal{O}^q_{\mathsf{cnm}}((c_0, c_1), (\mathsf{T}_0, \mathsf{T}_1))}$:
$(c'_0, c'_1) = (\mathsf{T}_0(c_0), \mathsf{T}_1(c_1))$
If $(c'_0, c'_1) = (c_0, c_1)$ return $\mathsf{same}^\star$
If $\text{Decode}(\Omega, (c'_0, c'_1)) = \perp$, return $\perp$ and "self-destruct"
Else return $(c'_0, c'_1)$.

Essentially, the oracle can be queried up to $q$ times with input functions $\mathsf{T}_0, \mathsf{T}_1 : \{0,1\}^n \to \{0,1\}^n$ and returns either $\mathsf{same}^\star$ (in case $(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1)) = (c_0, c_1)$), or $\perp$ (in case $\text{Decode}(\Omega, (\mathsf{T}_0(c_0), \mathsf{T}_1(c_1))) = \perp$), or $(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1))$ in all other cases. The only additional restriction is that in case where $\perp$ is returned the oracle "self-destructs" and answers all further queries with $\perp$. Furthermore, in the construction of [17] the adversary has access to leakage oracles $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_0, \cdot)$, $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_1, \cdot)$, that can be queried to retrieve up to $\mathsf{lb}_{\mathsf{code}}$ bits of information on each half of the target encoding. The

access to the leakage oracles will be useful in our setting to obtain continuous leakage resilience on the buses.

Below is a formal definition of continuous non-malleable leakage resilient (CNMLR) codes.

**Definition 1** (CNMLR code)**.** *Let* $\mathcal{C} = (\text{Init}, \text{Encode}, \text{Decode})$ *be an encoding scheme. For any adversary* $\mathcal{A}$ *consider the following interactive game for a uniform bit* $b \in \{0,1\}$*:*

$\underline{\text{GAME}^{\mathsf{cnmlr},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(b)}$

Compute $\Omega \leftarrow \text{Init}(1^k)$ and give it to $\mathcal{A}$.
Receive $(x_0, x_1)$ from $\mathcal{A}$ with $|x_0| = |x_1|$.
Compute $(c_0, c_1) \leftarrow \text{Encode}(x_b)$.
Compute a bit $b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_0,\cdot),\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_1,\cdot),\mathcal{O}^q_{\mathsf{cnm}}((c_0,c_1),(\cdot,\cdot))}$.
Output $b'$.

*We say that* $\mathcal{C}$ *is* $q$*-continuously non-malleable* $\mathsf{lb}_{\mathsf{code}}$*-leakage resilient (*$(\mathsf{lb}_{\mathsf{code}}, q)$*-CNMLR in short), if for all PPT adversaries* $\mathcal{A}$ *the following holds:*

$$\Pr[\text{GAME}^{\mathsf{cnmlr},q,\mathsf{lb}_{code}}_{\mathcal{C},\mathcal{A}}(1) = 1] - \Pr[\text{GAME}^{\mathsf{cnmlr},q,\mathsf{lb}_{code}}_{\mathcal{C},\mathcal{A}}(0) = 1] \leq negl(k) \ .$$

We remark that depending on the actual code the public parameters $\Omega$ can be empty. However, whenever present, they are assumed to be untamperable. This corresponds, e.g., to the assumption that the common reference string cannot be modified in the construction of [17].

**Composability.** We consider a strengthening of Definition 1, where the adversary is allowed to tamper with a vector of codewords. Let $\mathbf{x} = (x_1, \ldots, x_m) \in (\{0,1\}^*)^m$ be a vector, and define the following oracle $\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0, \mathbf{c}_1))$. The oracle is parametrized by $m \times n$ matrices $(\mathbf{c}_0, \mathbf{c}_1)$ such that $\mathbf{c}_0 = (c_0^1, \ldots, c_0^m)$, $\mathbf{c}_1 = (c_1^1, \ldots, c_1^m)$ where $(c_0^i, c_1^i) = \text{Encode}(\Omega, x_i)$ (i.e., the $i$-th row of $\mathbf{c}_b$ is equal to $c_b^i$). Furthermore, let $mn := |\mathbf{c}_b|$ denote the bit length of $\mathbf{c}_b$; then the oracle takes as input functions $\mathsf{T}_0, \mathsf{T}_1 : \{0,1\}^{mn} \to \{0,1\}^n$.[7]

$\underline{\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0, \mathbf{c}_1), (\mathsf{T}_0, \mathsf{T}_1))}$:

$(c_0', c_1') = (\mathsf{T}_0(\mathbf{c}_0), \mathsf{T}_1(\mathbf{c}_1))$
If $\exists\ i \in [m]$ such that $(c_0', c_1') = (c_0^i, c_1^i)$ return $(\mathsf{same}^\star, i)$
If $\text{Decode}(\Omega, (c_0', c_1')) = \perp$, return $\perp$ and "self-destruct"
Else return $(c_0', c_1')$.

We also consider a leakage oracle $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c})$ which allows leakage from a vector of values. It limits the possible leakage from each individual value to be less than $\mathsf{lb}_{\mathsf{code}}$ bits. It takes as input an $m$-dimensional vector $\mathbf{c}$ and a set $S \subset [m]$ specifying which elements to leak from, along with a leakage function $\mathsf{L} : \{0,1\}^* \to \{0,1\}^\lambda$ for some $\lambda$. It keeps a state what is the current amount of information $(\lambda_1, \ldots, \lambda_m)$ that has been leaked. Initially, we set $(\lambda_1, \ldots, \lambda_m) = (0, \ldots, 0)$.

[7] The fact that the tampering function can output a single codeword, instead of $m$, might seem odd at a first look. However, this variant is sufficient for our purpose. Moreover, it is easy to see that the more general setting where the tampering functions can output $m$ codewords, can be emulated by accessing the above oracle $\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0, \mathbf{c}_1))$ for $m$ times.

$\underline{\mathcal{O}^{\mathsf{lb}}_{\mathsf{code}}(\mathbf{c},(S,\mathsf{L}))}$:

Compute $L \leftarrow \mathsf{L}\{\mathbf{c}[i]\}_{i\in S})$ and let $\lambda = |L|$
For $i \in S$ update $\lambda_i \leftarrow \lambda_i + \lambda$
If $\lambda_i < \mathsf{lb}_{\mathsf{code}}$ for $i = 1, \ldots, m$, then return $L$
Else return $\perp$.

Using the above two oracles, we can now now define our notion of adaptive composable CNMLR codes. Besides being composable our notion also is adaptive in the sense that *after* the adversary interacted with the oracle he can specify new messages that he would like to append to the set of encodings.

**Definition 2** (Adaptive composability of CNMLR)**.** *Let* $\mathcal{C} = (\text{Init}, \text{Encode}, \text{Decode})$ *be a* $(\mathsf{lb}_{\mathsf{code}}, q)$*-CNMLR encoding scheme. For some adversary* $\mathcal{A}$ *consider the following interactive game:*

$\underline{\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(b)}$

Compute $\Omega \leftarrow \text{Init}(1^k)$ and obtain $(x_0^1, x_1^1) \leftarrow \mathcal{A}(\Omega)$. Set $\mathbf{c}_0 = \emptyset, \mathbf{c}_1 = \emptyset$
For $i = 1, \ldots, m$, do the following:
  Compute $(c_0^i, c_1^i) \leftarrow \text{Encode}(\Omega, x_b^i)$ and update $(\mathbf{c}_0, \mathbf{c}_1)$ with $((\mathbf{c}_0, c_0^i), (\mathbf{c}_1, c_1^i))$.
  Receive $(x_0^{i+1}, x_1^{i+1}) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$, with $|x_0^{i+1}| = |x_1^{i+1}|$.
Receive a bit $b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$.
Output $b'$.

*We say that* $\mathcal{C}$ *is adaptively* $m$*-composable if for all PPT adversary* $\mathcal{A}$ *the following holds:*

$$\Pr[\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{code}}_{\mathcal{C},\mathcal{A}}(1) = 1] - \Pr[\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{code}}_{\mathcal{C},\mathcal{A}}(0) = 1] \leq negl(k) \ .$$

Notice that in each iteration of the loop the domain of the tampering functions that the adversary submits to the $\mathcal{O}^q_{\mathsf{cnm}}$ oracle changes. In particular, in the $i$-th iteration the domain of the functions $\mathsf{T}_0, \mathsf{T}_1$ is $\{0,1\}^{(i-1)n}$.

We argue that Definition 2 and Definition 1 are equivalent (asymptotically). Clearly, adaptive composability implies continuous non-malleability for $m = 1$. The other direction follows by the theorem below (whose proof is given in Appendix A):

**Theorem 1.** *Let* $\mathcal{C} = (\text{Init}, \text{Encode}, \text{Decode})$ *be a* $(\mathsf{lb}_{\mathsf{code}}, q)$*-CNMLR code, then* $\mathcal{C}$ *is also adaptively* $m$*-composable for any polynomial* $m = poly(k)$*.*

# 3 Leakage and Tamper Resilient RAM

Our RAM architecture can implement some keyed functionality $\mathcal{G}_{\mathsf{K}}$, e.g., an AES running with key $\mathsf{K}$ taking as input messages and producing the corresponding ciphertexts. The RAM has one or more independently tamperable and leaky secret storages, one tamperable public storage[8] and one CPU. There is a leak-free and tamper-free pre-processing phase, which outputs an encoding of the functionality $\mathcal{G}_{\mathsf{K}}$. Notice that we do not only encode the key but also the description of the "program code" describing $\mathcal{G}_{\mathsf{K}}$. This will enable us to allow tampering *even* with the functionality itself. Looking ahead, our encoding of $\mathcal{G}_{\mathsf{K}}$ ensures that if the adversary attempts to change the

[8]The reason for having also a public disk is that we need a public place to put the program for the CPU, such that our emulator can learn the structure of the compiled program.

program code he always also tampers with $\mathsf{K}$. This is achieved by tying together the encodings of the key and the program code.

The initial encoding consists of secret part(s) (containing data) which we store in the secret disk(s) and a public part (containing instructions) which is stored on the public disk. The input and output of the function that can be chosen by the user of the RAM is stored in some *specific* locations on the public disk (say, right after the program). We allow the exact location of the input and output parameters to be *program specific*, but assume that access to the public disk allows to efficiently determine the input and output (in case the public disk was not tampered). In the online phase, the CPU loads an instruction from the public disk and data from the secret disk(s) (as specified by the instruction). Reading from the public disk might involve reading part of the input. Then it computes and stores back the intermediate results on the secret disk(s) and the public disk and processes the next instruction. The next instruction is found on the public disk at the location given by a program counter $\mathtt{pc}$ which incremented by one in each invocation of the CPU and which is reset when CPU raise a flag $\mathtt{T} = 1$). Writing to the public disk could involve writing part of the output. The adversary is allowed to tamper with the disks between each two invocations of the CPU; furthermore the adversary is allowed to leak (independently) from the secret disk(s) and from the buses carrying the information between the CPU and the secret disks. As the size of the CPU is independent of the size of the functionality $\mathcal{G}_{\mathsf{K}}$, and hence it cannot evaluate the complete functionality, our model allows the adversary to tamper arbitrary many times with the state of the RAM during the execution of $\mathcal{G}_{\mathsf{K}}$.

In the following, we give a formal presentation of our model. We begin with some basic definitions that formalize the notion of a program and its instructions.

**Definition 3** (Dimension of storage, instructions and programs)**.** *Let $\ell, L, \tau, d, C, P \in \mathbb{N}$ be some parameters explained below. We say the* dimension of a storage $S$ *is $\ell \times L$ if the storage contains $L$ words and each word is an $\ell$-bit string. Each word is located by a unique identifier in $\{1, \ldots, L\}$. An $(\tau, L, d, C)$-bound* instruction $\mathcal{I}$ *is defined as a triple $(\mathsf{Y}, \mathsf{I}, \mathsf{O})$ where, $\mathsf{Y} \in \{0,1\}^{\tau}$, and $\mathsf{I}, \mathsf{O} \in (\{0, \ldots, C-1\} \times \{1, \ldots, L\})^d$. A* program $\mathcal{P}$ *of length $P$ is defined as a $P$-tuple of instructions $(\mathcal{I}_1, \ldots, \mathcal{I}_P)$.*

One may think of $\mathsf{Y}$ as the type of operation (e.g., a NAND operation) that is computed by the instruction. The $d$-tuples $\mathsf{I}, \mathsf{O}$ define the position on the disks where to read the inputs and where to write the outputs of the instruction. Here, $C$ is a bound on the number of the disks and $L$ is a bound for the number of positions on the disk. We next describe our RAM architecture generically for the case of $C$ disks (we will require RAMs with 2 and 3 disks for the remainder of the paper).

**Specification of RAM:** A Random Access Machine (RAM in short) $\mathbf{R}$ is specified by $\mathbf{R} = (\tau, \ell, L, d, C, p, \mathsf{Random}, \mathsf{Compute})$ and consists of

1. $(\mathcal{SD}_1, \mathcal{SD}_2, \ldots, \mathcal{SD}_{C-1})$ : $C-1$ secret storages, each of dimension $\ell \times L$
2. $\mathcal{PD}$ : The public storage of dimension $\ell \times p$.
3. $\mathsf{CPU}$ : A procedure which is formally written as pseudo-code in Algorithm 1. The $\mathsf{CPU}$ is connected to the different disks by buses $\mathsf{Bs}_j$, which are used to load and store data. It has $2d+1$ internal temporary registers: $d+1$ input registers $(\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_d)$ and $d$ output registers $(\mathtt{O}_1, \ldots, \mathtt{O}_d)$; each register can store $\ell$ bits. $\mathsf{CPU}$ takes as inputs data send through the buses,

**Algorithm 1** CPU

**Input**: $(\mathtt{pc}, \mathtt{ac}, \mathcal{PD}, \mathcal{SD}_1, \mathcal{SD}_2, \ldots, \mathcal{SD}_{C-1}, \mathsf{Leak}_1, \ldots, \mathsf{Leak}_{C-1})$
Let $D_0 = \mathcal{PD}; D_1 = \mathcal{SD}_1; \ldots; D_{C-1} = \mathcal{SD}_{C-1}$ *// Loading...*
If $D_0[\mathtt{pc}]$ is not of the form $(\mathsf{Y}, \mathsf{I}, \mathsf{O})$ then output

$$((D_0, \ldots, D_{C-1}), \mathtt{B} = 1, \mathtt{T} = 0, ()) \qquad \textit{// Self-destruct}$$

Let $(\mathsf{Y}, \mathsf{I}, \mathsf{O}) = D_0[\mathtt{pc}]$
For $j = 1, \ldots, C-1$ initialize the bus $\mathsf{Bs}_j = ()$ *// Clear the buses*
Load $\mathtt{R}_0 \leftarrow (\mathsf{Y}, \mathsf{I}, \mathsf{O})$
**for** $j = 1 \to d$ **do**
  Let $(\mathtt{src}_j, \mathtt{loc}_j) = \mathsf{I}[j]$ *// Load input from disk* $\mathtt{src}_j$ *at position* $\mathtt{loc}_j$
  Load $\mathtt{R}_j \leftarrow D_{\mathtt{src}_j}[\mathtt{loc}_j]$
  If $\mathtt{src}_j > 0$, set $\mathsf{Bs}_{\mathtt{src}_j} \leftarrow (\mathsf{Bs}_{\mathtt{src}_j}, \mathtt{R}_j)$ *// Write data from secret disks to buses*
**end for**
  *// Computing...*
Sample $r \leftarrow \mathsf{Random}$
Compute $((\mathtt{O}_1, \ldots, \mathtt{O}_d), \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{Compute}((\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_d), r, \mathtt{pc}, \mathtt{ac})$
  *// Storing...*
**for** $j = 1 \to d$ **do**
  Let $(\mathtt{tar}_j, \mathtt{loc}_j) = \mathsf{O}[j]$
  Store $D_{\mathtt{tar}_j}[\mathtt{loc}_j] \leftarrow \mathtt{O}_j$ *// Store output on disk* $\mathtt{src}_j$ *at position* $\mathtt{loc}_j$
  If $\mathtt{tar}_j > 0$, set $\mathsf{Bs}_{\mathtt{tar}_j} \leftarrow (\mathsf{Bs}_{\mathtt{tar}_j}, \mathtt{O}_j)$
**end for**
For $j = 1, \ldots, C-1$, let $\lambda_j = \mathsf{Leak}_j(\mathsf{Bs}_j)$ *// Compute leakage from the buses*
**Output**: $((D_0, \ldots, D_{C-1}), \mathtt{B}, \mathtt{T}, (\lambda_1, \ldots, \lambda_{C-1}))$

a strictly increasing activation[9] counter $\mathtt{ac}$, a program counter $\mathtt{pc}$ which is strictly increasing within one activation and reset between activations. The $\mathsf{CPU}$ runs in three steps: (i) $d$ loads, (ii) 1 computation and (iii) $d$ stores. In the computation step $\mathsf{CPU}$ calls $\mathsf{Random}$ and $\mathsf{Compute}$ to generate fresh randomness and evaluate the instruction.

(a) $\mathsf{Random}$: This algorithm is used to sample randomness $r$.

(b) $\mathsf{Compute}$: This algorithm will evaluate one particular instruction. To this end, it takes data from the temporary registers $(\mathtt{R}_0, \ldots, \mathtt{R}_d)$, the counters $\mathtt{ac}, \mathtt{pc}$ and the randomness $r \leftarrow \mathsf{Random}$ as input and outputs the data to be stored into the output registers $(\mathtt{O}_1, \ldots, \mathtt{O}_d)$, the self-destruct indicator bit $\mathtt{B}$ which indicates if $\mathsf{CPU}$ needs to stop execution, and the completion indicator bit $\mathtt{T}$ which indicates the completion of the current activation.

$\mathsf{CPU}$ outputs the possibly updated storages $(\mathcal{PD}, \mathcal{SD}_1, \mathcal{SD}_2, \ldots, \mathcal{SD}_{C-1})$, the self-destruct indicator ($\mathtt{B}$) and the completion indicator ($\mathtt{T}$). Notice that the CPU does not need to take $\mathtt{B}$

[9] We call the time in which the RAM computes the output $\mathcal{G}_{\mathsf{K}}(x)$ for single $x$ one activation, and the time in which the procedure $\mathsf{CPU}$ is run once, one execution.

and $\mathtt{T}$ as input as these bits are only written.

Running the RAM involves iteratively executing the CPU. In between executions of the CPU we increment $\mathtt{pc}$. When the CPU return $\mathsf{T} = 1$ we reset $\mathtt{pc} = 0$ and increment the activation counter $\mathtt{ac}$. When the CPU return $\mathsf{B} = 1$ the CPU self destruct. After this no more execution of the CPU takes place.

Note that our RAM does not explicitly allow indirection, as in loading e.g. $D_1[D_1[127]]$. It would have to do this in two steps: load $D_1[127]$ and store it on public disk as an input location as part of the next instruction, and then have this instruction executed. This is to force that the access pattern is leaked, as we do not want to assume that our RAM can hide the access pattern.

### 3.1 The RAM Compiler

From now on we consider a specific architecture where the RAM has two secret disks $\mathcal{SD}_1$ and $\mathcal{SD}_2$ and one public disk $\mathcal{PD}$. Informally, a RAM compiler $\mathbf{C}$ takes as input the description of a functionality $\mathcal{G}$ with secret key $\mathsf{K}$, and outputs an encoding of the functionality itself, to be executed trough a RAM $\mathbf{R}$.

**Definition 4** (RAM-Compiler)**.** *An $(\ell, L, p)$-RAM-Compiler $\mathbf{C}$ is a PPT algorithm which takes a keyed-function description $\mathcal{G}$ and a key $\mathsf{K} \in \{0,1\}^*$ as input, and outputs an encoding of the form $((I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}), \sigma_0, \sigma_1, \sigma_2)$ such that $\sigma_0$ can be stored into a disk of dimension $\ell \times p$ and $\sigma_1, \sigma_2$ can be stored into disks each of dimension $\ell \times L$.*

Here $I$ is the position where the input is put on the public disk, $\ell_I$ is the length of the input, $O$ is the position where the output is put on the public disk, and $\ell_O$ is the length of the output. The mappings $\mathcal{X}, \mathcal{Y}$ are used to parse the inputs (resp., the outputs) of the RAM as a certain number of words of length $\ell$ (resp., as a value in the range of $\mathcal{G}_{\mathsf{K}}$).

We define a RAM-scheme $\mathsf{RS}$ as the ordered pair $(\mathbf{C}, \mathbf{R})$ such that $\mathbf{C}(\mathcal{G}_{\mathsf{K}})$ is supposed to be executed in $\mathbf{R}$. Below we define what it means for a RAM-scheme $\mathsf{RS} = (\mathbf{C}, \mathbf{R})$ to be correct. Informally, the definition says that for any tuple of inputs $(x_1, \ldots, x_N)$ the execution of the RAM $\mathbf{R}$ and the evaluation of the function $\mathcal{G}_{\mathsf{K}}$ have identical output distribution except with negligible probability. This is formalized in the definition below.

**Definition 5** (Correctness of RAM-Scheme)**.** *A RAM-Scheme $\mathsf{RS}$ is defined as an ordered pair $\mathsf{RS} = (\mathbf{C}, \mathbf{R})$ where $\mathbf{C}$ is an $(\ell, L, p)$-RAM-Compiler and $\mathbf{R}$ is a $(\tau, \ell, L, d, 3, p, \mathsf{Random}, \mathsf{Compute})$-RAM. We say the RAM-scheme $\mathsf{RS}$ is correct if for any function $\mathcal{G}$, any key $\mathsf{K} \in \{0,1\}^*$ and any vector of inputs $(x_1, \ldots, x_N)$ it holds that $\Pr[\textsc{Game}^{\mathsf{Real}}_{\mathsf{hon}}(x_1, \ldots, x_N) = 0] \leq negl(k)$, where the experiment $\textsc{Game}^{\mathsf{Real}}_{\mathsf{hon}}(x_1, \ldots, x_N)$ is defined as follows:*

$\textsc{Game}^{\mathsf{Real}}_{\mathsf{hon}}(x_1, \ldots, x_N)$**:** Initialize all the public values $\mathtt{T}, \mathtt{B}, \mathtt{ac}, \mathtt{pc}$ to 0. Run the compiler $\mathbf{C}$ on $(\mathcal{G}, \mathsf{K})$ to generate the encoding $((I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}), \sigma_0, \sigma_1, \sigma_2) \leftarrow \mathbf{C}(\mathcal{G}, \mathsf{K})$, and store it into the disks of $\mathbf{R}$ as follows: $\mathcal{PD} \leftarrow \sigma_0$, $\mathcal{SD}_1 \leftarrow \sigma_1$ and $\mathcal{SD}_2 \leftarrow \sigma_2$. For $i = 1 \rightarrow N$ do as follows. Encode the input $(x_{i,0}, \ldots, x_{i,\ell_I - 1}) \leftarrow \mathcal{X}(x_i)$, store it on the public disk $\mathcal{PD}[I + j] \leftarrow x_{i,j}$ (for $0 \leq j < \ell_I$) and run the following activation loop:

1. Run $\mathsf{CPU}$ and update the disks $(\mathcal{PD}, \mathcal{SD}_1, \mathcal{SD}_2, \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{CPU}(\mathtt{pc}, \mathtt{ac}, \mathcal{PD}, \mathcal{SD}_1, \mathcal{SD}_2)$.[10]

[10] When we do not specify leakage functions, we assume that they are all the constant function outputting the empty string, and we ignore the leakage in the output vector.

2. If $\mathtt{B} = 1$ return 0.

3. If $\mathtt{T} = 1$ then do as follows:

   (a) Let $y_i \leftarrow \mathcal{Y}(\mathcal{PD}[O], \ldots, \mathcal{PD}[O + \ell_O - 1])$. If $y_i \neq \mathcal{G}_{\mathsf{K}}(x_i)$ then return 0 [11].

   (b) Otherwise, increment the activation counter $\mathtt{ac} \leftarrow \mathtt{ac} + 1$. If $\mathtt{ac} = N$, then return 1 and reset the program counter $\mathtt{pc} \leftarrow 0$. Exit the activation loop. Otherwise increment the program counter $\mathtt{pc} \leftarrow \mathtt{pc} + 1$ and continue the activation loop.

## 3.2 The Security Model

We now proceed to define security of a RAM-Scheme, using the real-ideal world paradigm. In the following we let $k$ denote the security parameter.

### 3.2.1 Real Execution $\mathbf{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)$

Consider a RAM-scheme $\mathsf{RS} = (\mathbf{C}, \mathbf{R})$. First it runs $\mathbf{C}$ which takes the description of $\mathcal{G}$ and a key $\mathsf{K}$ as inputs and generates encoding of the form $((I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}), \sigma_0, \sigma_1, \sigma_2)$ such that $(\sigma_0, \sigma_1, \sigma_2)$ are stored into disks $(\mathcal{PD}, \mathcal{SD}_1, \mathcal{SD}_2)$ of $\mathbf{R}$ respectively. Then it advances to the online phase when the adversary $\mathcal{A}$ gets read/write access to $\mathcal{PD}$, which allows $\mathcal{A}$ to run $\mathbf{R}$ on inputs of his choice. Moreover, $\mathcal{A}$ can tamper with $\mathcal{SD}_1, \mathcal{SD}_2$ independently between each execution of the $\mathsf{CPU}$. Recall that our instructions are universal and independent of the size of $\mathcal{G}$, which allows $\mathcal{A}$ to tamper arbitrary many times with $\mathcal{SD}_1, \mathcal{SD}_2$ during an evaluation of $\mathcal{G}_{\mathsf{K}}(\cdot)$. It gets leakages from disks $\mathcal{SD}_1, \mathcal{SD}_2$—where the total leakage from each secret disk is bounded by $\mathsf{lb}_{\mathtt{disk}}$ bits—and buses[12] $\mathsf{Bs}_1, \mathsf{Bs}_2$—where the total leakages from each bus is bounded by $\mathsf{lb}_{\mathtt{bus}}$. The procedure $\mathsf{CPU}$ is leakage and tamper proof. See Fig. 1 for a formal description.

A few remarks corresponding to the description are in order.

1. **Adaptivity.** We stress that by writing the public disk, the adversary is allowed to query the RAM on adaptively chosen inputs. Also note that the adversary can issue tampering commands to the public disk and the two secret disks independently. Since $\mathcal{PD}$ is public, this always allows the adversary to tamper with $\mathcal{SD}_1$ (or $\mathcal{SD}_2$) using hard-wired values from the public disk.

2. **Tampering within executions.** Notice that without loss of generality we can assume that the adversary does not tamper within the execution of the CPU. Since the instructions are stored in the public disk $\mathcal{PD}$ the adversary knows the exact sequence of the locations to be read by the CPU and hence equivalently the adversary can just load some location, tamper and then execute before loading the next one. Essentially this is done by prohibiting any *indirect* instruction such that e.g. there is an instruction in some location in public disk which points to the 5-th location on the secret disk $\mathcal{SD}_1$ and that 5-th location on $\mathcal{SD}_1$ in turn points to the 8-th location on $\mathcal{SD}_2$. In that case it would have been impossible to predict the whole sequence of locations supposed to be read by the CPU which is undesirable.

[11] Throughout the paper we, in general, denote the name of the disks by $\mathcal{PD}, \mathcal{SD}_1, \mathcal{SD}_2, \ldots$ and the respective contents by $D_0, D_1, \ldots$, however, sometimes we abuse notations and use them interchangeably.

[12] Buses are the channels modelling the communication between the CPU and the secret disks.

In the following we let $D_0 := \mathcal{PD}, D_1 := \mathcal{SD}_1, D_2 := \mathcal{SD}_2$ and $i \in \{0,1,2\}$ identifies a disk.

1. Initialization: Sample the key $\mathsf{K}$ according to the distribution needed by the primitive. Initialize the activation counter $\mathtt{ac} \leftarrow 0$, the program counter $\mathtt{pc} \leftarrow 0$, the self-destruct bit $\mathtt{B} \leftarrow 0$, the activation indicator $\mathtt{T} \leftarrow 0$ and the total leakage count for each disk: $\mathtt{lt}_i \leftarrow 0$ for $i \in \{0,1,2\}$.
2. Pre-processing: Sample an encoding by running the compiler $(P, \sigma_0, \sigma_1, \sigma_2) \leftarrow \mathbf{C}(\mathcal{G}, \mathsf{K})$ (where $P = (I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y})$). Store the encoding into disks: $D_0 \leftarrow \sigma_0$, $D_1 \leftarrow \sigma_1$, $D_2 \leftarrow \sigma_2$. Give $P$ to $\mathcal{A}$.
3. Online: Get command $\mathtt{CMD}$ from $\mathcal{A}$ and act as follows according to the command-type:
   (a) If $\mathtt{CMD} = (\mathtt{STOP}, \mathsf{O}_{\mathtt{real}})$ then return $\mathsf{O}_{\mathtt{real}}$ and halt.
   (b) If $\mathtt{CMD} = (\mathtt{LEAK}, i, \mathsf{Leak}(\cdot))$, proceed as follows. If $i = 0$ then compute $\lambda \leftarrow \mathsf{Leak}(D_i)$ and give $\lambda$ to $\mathcal{A}$. Otherwise, update total leakage, $\mathtt{lt}_i \leftarrow \mathtt{lt}_i + |\mathsf{Leak}|$, and if $\mathtt{lt}_i \leq \mathsf{lb}_{\mathtt{disk}}$ then compute $\lambda \leftarrow \mathsf{Leak}(D_i)$ and give $\lambda$ to $\mathcal{A}$.
   (c) If $\mathtt{CMD} = (\mathtt{TAMPER}, i, \mathsf{Tamper}(\cdot))$ then modify $D_i$ using the tampering function: $D_i \leftarrow \mathsf{Tamper}(D_i)$.
   (d) If $\mathtt{CMD} = (\mathtt{EXEC}, \mathsf{Leak}_1, \mathsf{Leak}_2)$ and $\mathtt{B} = 0$ then ignore the command if $|\mathsf{Leak}_1| > \mathsf{lb}_{\mathtt{bus}}$ or $|\mathsf{Leak}_2| > \mathsf{lb}_{\mathtt{bus}}$, and otherwise proceed as follows:
      i. Run $\mathsf{CPU}$ and replace the existing disks by the modified output: $(D_0, D_1, D_2, \mathtt{B}, \mathtt{T}, \Lambda) \leftarrow \mathsf{CPU}(\mathtt{pc}, \mathtt{ac}, D_0, D_1, D_2)$.
      ii. Give $(\mathtt{T}, \Lambda)$ to $\mathcal{A}$.
      iii. Check the completion of current activation: If $\mathtt{T} = 1$ then start a new activation by incrementing the activation counter: $\mathtt{ac} \leftarrow \mathtt{ac} + 1$ and re-initializing the program counter: $\mathtt{pc} \leftarrow 0$
      iv. Increment the program counter: $\mathtt{pc} \leftarrow \mathtt{pc} + 1$ and go to Step-3.

**Figure 1:** Real Execution $\text{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)$

3. **Implicitly handling input/output.** In the above we do not explicitly handle input/output because it is taken care of in the tamper query (cf. Step-3(c)) in which the adversary can place the input on some specific location in the public disk and read the output from that as well. Also we do not give the public disk explicitly to $\mathcal{A}$ as it anyway can access it fully by the leakage query, which is unrestricted in this case (in Step-3(b) there is no check for $i = 0$).

### 3.2.2 Hybrid Execution $\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$

We propose a hybrid model which is actually an *intermediate* model between the real and the ideal one (to be presented next). In the hybrid model there is also a compiler denoted by $\mathbf{C}_{\mathsf{h}}$, which compiles a keyed-function to some encoding having a particular form to be executed in a hybrid-RAM denoted by $\mathbf{R}_{\mathsf{h}}$ which has a public disk $\mathcal{PD}^{\mathsf{h}}$ and a CPU also, but only one secret disk $\mathcal{SD}^{\mathsf{h}}_1$. Importantly, in this model the secret disk is leak-free but "limitedly" tamperable such that the adversary can only *copy* values within the disk and/or *replace* elements with a chosen value. The hybrid-compiler is defined in the same way as the RAM-compiler (c.f. Definition 4) with adequate modifications.

**Definition 6** (Hybrid Compiler)**.** *An $(\ell, L, p)$-Hybrid-Compiler $\mathbf{C}_{\mathsf{h}}$ is a PPT algorithm which*

*takes a keyed-function description $\mathcal{G}$ and a key $\mathsf{K}$ as input and outputs an encoding of the form $((I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}), \omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}})$ such that $\omega_0^{\mathsf{h}}$ can be stored into a disk of dimension $\ell \times p$ and $\omega_1^{\mathsf{h}}$ can be stored into a disk of dimension $\ell \times L$.*

Similar to the RAM-scheme we define a hybrid scheme $\mathsf{HS}$ as the ordered pair $(\mathbf{C}_{\mathsf{h}}, \mathbf{R}_{\mathsf{h}})$ such that $\mathbf{C}_{\mathsf{h}}(\mathcal{G}_{\mathsf{K}})$ is supposed to be executed in $\mathbf{R}_{\mathsf{h}}$. The definition below specifies correctness of a hybrid-scheme and is similar to Definition 5.

**Definition 7** (Correctness of Hybrid-Scheme)**.** *A Hybrid-Scheme $\mathsf{HS}$ is defined as an ordered pair $\mathsf{HS} = (\mathbf{C}_{\mathsf{h}}, \mathbf{R}_{\mathsf{h}})$ where $\mathbf{C}_{\mathsf{h}}$ is an $(\ell, L, p)$-Hybrid-Compiler and $\mathbf{R}_{\mathsf{h}}$ is a $(\tau, \ell, L, d, 2, p, \mathsf{Random}, \mathsf{Compute})$-RAM. We say the hybrid-scheme $\mathsf{HS}$ is correct if for any function $\mathcal{G}$, any key $\mathsf{K}$ and any vector of inputs $(x_1, \ldots, x_N)$ it holds that $\Pr[\textsc{Game}_{\mathsf{hon}}^{\mathsf{Hyb}}(x_1, \ldots, x_N) = 0] \leq negl(k)$, where the experiment $\textsc{Game}_{\mathsf{hon}}^{\mathsf{Hyb}}(x_1, \ldots, x_N)$ is defined as follows:*

**$\textsc{Game}_{\mathsf{hon}}^{\mathsf{Hyb}}(x_1, \ldots, x_N)$:** Initialize all the public values $\mathtt{T}, \mathtt{B}, \mathtt{ac}, \mathtt{pc}$ to 0. Run the compiler $\mathbf{C}_{\mathsf{h}}$ on $(\mathcal{G}, \mathsf{K})$ to generate the encoding $((I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}), \omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}}) \leftarrow \mathbf{C}_{\mathsf{h}}(\mathcal{G}, \mathsf{K})$, and store it into the disks of $\mathbf{R}_{\mathsf{h}}$ as follows: $\mathcal{PD}^{\mathsf{h}} \leftarrow \omega_0^{\mathsf{h}}$ and $\mathcal{SD}_1^{\mathsf{h}} \leftarrow \omega_1^{\mathsf{h}}$. For $i = 1 \rightarrow N$ do as follows. Encode the input $(x_{i,0}, \ldots, x_{i,\ell_I - 1}) \leftarrow \mathcal{X}(x_i)$, store it on the public disk $\mathcal{PD}^{\mathsf{h}}[I + j] \leftarrow x_{i,j}$ (for $0 \leq j < \ell_I$), and run the following activation loop:

1. Run $\mathsf{CPU}^{\mathsf{h}}$ and update the disks $(\mathcal{PD}^{\mathsf{h}}, \mathcal{SD}_1^{\mathsf{h}}, \mathcal{SD}_2^{\mathsf{h}}, \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{CPU}^{\mathsf{h}}(\mathtt{pc}, \mathtt{ac}, \mathcal{PD}^{\mathsf{h}}, \mathcal{SD}_1^{\mathsf{h}}, \mathcal{SD}_2^{\mathsf{h}})$.
2. If $\mathtt{B} = 1$ return 0.
3. If $\mathtt{T} = 1$ then do as follows:
   (a) Let $y_i \leftarrow \mathcal{Y}(\mathcal{PD}^{\mathsf{h}}[O], \ldots, \mathcal{PD}^{\mathsf{h}}[O + \ell_O - 1])$. If $y_i \neq \mathcal{G}_{\mathsf{K}}(x_i)$ then return 0.
   (b) Otherwise, increment the activation counter $\mathtt{ac} \leftarrow \mathtt{ac} + 1$. If $\mathtt{ac} = N$, then return 1 otherwise reset the program counter $\mathtt{pc} \leftarrow 0$. Exit the activation loop. Otherwise, increment the program counter $\mathtt{pc} \leftarrow \mathtt{pc} + 1$ and continue the activation loop.

**Extra-space insensitivity.** We need a notion of the security of a scheme not depending too specifically on the size of the disks. This intuitively means that a hybrid scheme remains RAM-simulatable (c.f. Definition 10) even if we extend the disk space by any number of words. In our analysis, the hybrid-to-real emulator will leverage any such hybrid scheme, and our actual hybrid scheme (cf. Section 5) will have the property of being extra-space insensitive.

**Definition 8** (Extra-Space Insensitive)**.** *For a hybrid scheme $\mathsf{HS}$ and a non-negative integer $x$, let $\mathsf{HS}^{+x}$ be the same scheme as $\mathsf{HS}$, except that the disks have been extended by $x$ words, i.e., $\ell(\mathsf{HS}^{+x}) = \ell(\mathsf{HS}) + x$. We say that $\mathsf{HS}$ is extra-space insensitive if $\mathsf{HS}^{+x}$ is RAM-simulatable for $x \in \mathbb{N}$.*

**Hybrid execution.** Let $\mathsf{HS} = (\mathbf{C}_{\mathsf{h}}, \mathbf{R}_{\mathsf{h}})$ be a hybrid scheme. In the hybrid execution, we first run the Hybrid-Compiler $\mathbf{C}_{\mathsf{h}}$ which takes the description of the function $\mathcal{G}$ and a key $\mathsf{K}$ as inputs and generates encoding $(\omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}})$ which are stored into disks $(\mathcal{PD}^{\mathsf{h}}, \mathcal{SD}_1^{\mathsf{h}})$ of $\mathbf{R}_{\mathsf{h}}$ respectively. Then it advances to the online phase, where the adversary $\mathcal{B}$ gets full access to $\mathcal{PD}^{\mathsf{h}}$. Also it gets write-only access to $\mathcal{SD}_1^{\mathsf{h}}$ in a manner such that it can only (i) *copy* values *within* $\mathcal{SD}_1^{\mathsf{h}}$; (ii) *replace* some part of $\mathcal{SD}_1^{\mathsf{h}}$ or $\mathcal{PD}^{\mathsf{h}}$ with some chosen value.[13] The procedure $\mathsf{CPU}^{\mathsf{h}}$ is leakage and tamper proof.

[13]Notice that here the adversary $\mathcal{B}$ does not have any leakage from the secret disk $\mathcal{SD}_1^{\mathsf{h}}$.

In the following we let $D_0^{\mathsf{h}} := \mathcal{PD}^{\mathsf{h}}, D_1^{\mathsf{h}} := \mathcal{SD}_1^{\mathsf{h}}$ and $i \in \{0,1\}$ identifies a disk.

1. Initialization: Sample the key $\mathsf{K}$ according to the appropriate distribution (needed by the function). Initialize the activation counter $\mathtt{ac} \leftarrow 0$, the program counter $\mathtt{pc} \leftarrow 0$, the self-destruct bit $\mathtt{B} \leftarrow 0$ and the activation indicator bit $\mathtt{T} \leftarrow 0$. Additionally, for each memory position $j$ in $D_1^{\mathsf{h}}$ keep a value $\mathsf{S}[j] \in \perp \cup \mathbb{N}$, with the following definition: if $\mathsf{S}[j] = \perp$ then the value $D_1^{\mathsf{h}}[j]$ is known to the adversary. If $\mathsf{S}[j] = g \in \mathbb{N}$ then $D_1^{\mathsf{h}}[j]$ may not be known by the adversary and moreover $D_1^{\mathsf{h}}[j]$ is the $g$-th such secret value, numbered in the order of when it was written to the memory, in a sense made clear below. Initially $\mathsf{S}[j] = \perp$ for all $j$. Finally, initialize a record $C$ which in position $g$ keeps a counter $C[g]$ of how many times the $g$'th secret value was accessed. Initialize with $C[g] = 0$ for all $g$. Finally, initialize a counter keeping track of the index of the next secret value $\mathsf{ns} \leftarrow 0$.
2. Pre-processing: Sample an encoding by running the compiler $(P, \omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}}) \leftarrow \mathbf{C}_{\mathsf{h}}(\mathcal{G}, \mathsf{K})$ where $P = (I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y})$. Store the encoding into disks: $D_0^{\mathsf{h}} \leftarrow \omega_0^{\mathsf{h}}$, $D_1^{\mathsf{h}} \leftarrow \omega_1^{\mathsf{h}}$. Given $P$ to $\mathcal{B}$. While $\mathsf{ns} < |\omega_1^{\mathsf{h}}|$, let $\mathsf{S}[\mathsf{ns}] = \mathsf{ns}$ and update $\mathsf{ns} \leftarrow \mathsf{ns} + 1$.
3. Online: Get command $\mathtt{CMD}$ from $\mathcal{B}$ and act as follows according to the command-type:
   (a) If $\mathtt{CMD} = (\mathtt{STOP}, \mathsf{O}_{\mathsf{hyb}})$ then return $\mathsf{O}_{\mathsf{hyb}}$ and halt.
   (b) If $\mathtt{CMD} = (\mathtt{COPY}, (j, j'))$ then update $D_1^{\mathsf{h}}[j'] \leftarrow D_1^{\mathsf{h}}[j]$ and $\mathsf{S}[j'] \leftarrow \mathsf{S}[j]$.
   (c) If $\mathtt{CMD} = (\mathtt{REPLACE}, (i, j, \mathsf{val}))$ then update $D_i^{\mathsf{h}}[j] \leftarrow \mathsf{val}$. If $i = 1$, then update $\mathsf{S}[j] \leftarrow \perp$.
   (d) If $\mathtt{CMD} = (\mathtt{EXEC})$ and $\mathtt{B} = 0$ then do the following:
      i. Run $\mathsf{CPU}^{\mathsf{h}}$ and replace the existing disks by the modified output: $(D_0^{\mathsf{h}}, D_1^{\mathsf{h}}, \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{CPU}^{\mathsf{h}}(\mathtt{pc}, \mathtt{ac}, D_0^{\mathsf{h}}, D_1^{\mathsf{h}})$. Input $\mathtt{T}$ to $\mathcal{B}$.
      ii. Check the completion of the current activation: If $\mathtt{T} = 1$ then start a new activation by incrementing the activation counter: $\mathtt{ac} \leftarrow \mathtt{ac} + 1$ and re-initializing the program counter: $\mathtt{pc} \leftarrow 0$.
      iii. Increment the program counter: $\mathtt{pc} \leftarrow \mathtt{pc} + 1$.
      iv. Let $(\mathtt{src}_1, \mathtt{loc}_1), \ldots, (\mathtt{src}_d, \mathtt{loc}_d)$ be the memory positions read by the CPU. For all $i \in [d]$, where $\mathtt{src}_i = 1$ and $\mathsf{S}[\mathtt{loc}_i] \neq \perp$, update $C[\mathsf{S}[\mathtt{loc}_i]] \leftarrow C[\mathsf{S}[\mathtt{loc}_i]] + 1$.
      v. If there exists $i \in [d]$, where $\mathtt{src}_i = 1$ and $\mathsf{S}[\mathtt{loc}_i] \neq \perp$, then proceed as follows: Let $(\mathtt{tar}_1, \mathtt{loc}_1), \ldots, (\mathtt{tar}_d, \mathtt{loc}_d)$ be the memory positions written by the CPU. For $i \in [d]$, where $\mathtt{tar}_i = 1$, do $\mathsf{S}[\mathtt{loc}_i] = \mathsf{ns}$, $C[\mathsf{S}[\mathtt{loc}_i]] \leftarrow 1$, $\mathsf{ns} \leftarrow \mathsf{ns} + 1$. We call this case a *secret execution*.
      vi. If there does not exists $i \in [d]$, where $\mathtt{src}_i = 1$ and $\mathsf{S}[\mathtt{loc}_i] \neq \perp$, then proceed as follows: For $i \in [d]$, where $\mathtt{tar}_i = 1$, do $\mathsf{S}[\mathtt{loc}_i] \leftarrow \perp$. We call this case a *public execution*.
      vii. Go to Step-3.

**Figure 2:** Hybrid Execution ($\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$)

A complete description can be found in Fig. 2. Below, we elaborately explain the usage of the records used for book-keeping in Fig. 2. To handle the above *copy* and *replace* queries we maintain two storages namely $\mathsf{S}[j]$ and $C[j]$, which are indexed by a location $j$ on the secret disk. $\mathsf{S}[j]$ will keep track of whether a value stored at position $j$ on the secret disk is known to the adversary. If $\mathsf{S}[j] = \perp$ then the value $\mathcal{SD}_1^{\mathsf{h}}[j]$ is necessarily known by the adversary (e.g., this can happen when the adversary issues a *replace* query for position $j$ on the secret disk). If $\mathsf{S}[j] \neq \perp$, then $\mathcal{SD}_1^{\mathsf{h}}[j]$ may not be known by the adversary and the value of $\mathsf{S}[j]$ specifies "when" $\mathcal{SD}_1^{\mathsf{h}}[j]$ was written to

disk. For instance, initially $\mathsf{S}[j] = j$. On the other hand $C[j]$ is a counter which keeps track of the number of times the $j$-th location on the secret disk is accessed by the CPU. Sometimes we will use $C[\mathsf{S}[j]]$ to denote the number of times the value stored at position $j$ on $\mathcal{SD}_1^{\mathsf{h}}$ has been accessed. As there may be many copies of $\mathcal{SD}_1^{\mathsf{h}}[j]$ the storage $C$ is indexed by the value indexing the secret values and not the position on the disk. This is important to handle the leakage from the buses in the real model, because being accessed by the CPU in the hybrid execution is equivalent to leaking on that value in the real execution.

**$c$-Bounding Hybrid-scheme.** We say that a hybrid-scheme $\mathsf{HS}$ is $c$-bounding if for all $\mathcal{G}$ and all PPT adversaries $\mathcal{B}$ in $\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$ it holds that the scheme accesses each value on the secret disk at most $c$ times. More precisely, at any time during the execution and for all $j$ it holds that $C[j] \leq c$. To look ahead if $\mathsf{HS}$ is $c$-bounding then each secret data is touched at most $c$-times implying a bounded amount of leakage from the corresponding secrets.

### 3.2.3 Ideal Execution $\text{IDEAL}_{\mathcal{S},\mathcal{G}}(k)$

In the ideal execution, the ideal functionality for evaluating $\mathcal{G}$ interacts with the ideal adversary namely the simulator $\mathcal{S}$ as follows. First sample a key $\mathsf{K}$ and then repeat the following until a value was returned: Get a command from $\mathcal{S}$ and act differently according to the command-type.

1. If $\texttt{CMD} = (\texttt{STOP}, \mathsf{O}_{\texttt{ideal}})$, then return $\mathsf{O}_{\texttt{ideal}}$ and halt.
2. If $\texttt{CMD} = (\texttt{EVAL}, x)$, give $\mathcal{G}_{\mathsf{K}}(x)$ to $\mathcal{S}$.

### 3.2.4 RAM-Simulatability

We now specify what it means for a RAM-scheme to be RAM-simulatable.

**Definition 9** (RAM-Simulatability of RAM-Scheme)**.** *We say a RAM-scheme $\mathsf{RS}$ is RAM-simulatable against split-state tampering and $\mathsf{lb}$-leakage if for any function $\mathcal{G}$ and any PPT adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that*

$$\left\{\text{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} \approx_c \left\{\text{IDEAL}_{\mathcal{S},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}}$$

*where $\text{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)$ and $\text{IDEAL}_{\mathcal{S},\mathcal{G}}(k)$ denote the output distributions of the real execution and the ideal execution (as described above) respectively.*

RAM-simulatability will be proven via an intermediate step using the hybrid scheme. The following two definitions formalize this in a modular way.

**Definition 10** (RAM-Simulatability of Hybrid-Scheme)**.** *We say a hybrid-scheme $\mathsf{HS}$ is RAM-simulatable if for any function $\mathcal{G}$ and any PPT adversary $\mathcal{B}$ there exists a PPT simulator $\mathcal{S}$ such that*

$$\left\{\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} \approx_c \left\{\text{IDEAL}_{\mathcal{S},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}}$$

*where $\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$ and $\text{IDEAL}_{\mathcal{S},\mathcal{G}}(k)$ denote the output distributions of the real execution and the hybrid execution (as described above) respectively.*

**Definition 11** (Hybrid-to-Real Emulator)**.** *A $(c, \mathsf{lb}, q)$-bounded hybrid-to-real emulator $\mathcal{E}$ is defined as an efficient transformation which transforms any c-bounding, extra-space insensitive hybrid-scheme $\mathsf{HS}$ into a RAM-scheme $\mathsf{RS} = \mathcal{E}(\mathsf{HS})$ such that for any function $\mathcal{G}$ and any PPT adversary $\mathcal{A}$ there exists a PPT (hybrid) adversary $\mathcal{B}$ such that:*

$$\left\{\text{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} \approx_c \left\{\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}}$$

*where $\text{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)$ and $\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$ denote the output distributions of the real execution and the hybrid execution (as described above) respectively.*

Informally speaking, the above definition says that whatever the adversary $\mathcal{A}$ can do in the real world against the "transformed" RAM scheme $\mathsf{RS} = \mathcal{E}(\mathsf{HS})$, can be simulated by the hybrid adversary $\mathcal{B}$ in the hybrid world, where $\mathcal{B}$ has restricted power.

The following theorem is immediate.

**Theorem 2.** *Assume that $\mathcal{E}$ is a $(c, \mathsf{lb}, q)$-bounded hybrid-to-real emulator. Assume that $\mathsf{HS}$ is a RAM-simulatable hybrid scheme and also c-bounding and extra-space insensitive. Let $\mathsf{RS} = \mathcal{E}(\mathsf{HS})$. Then $\mathsf{RS}$ is RAM-simulatable against split-state q-tampering and $\mathsf{lb}$-leakage.*

We are now ready to state our main theorem. The proof follows by putting the following things together: (i) our construction of a hybrid-to-real emulator (cf. Theorem 4 in Section 4), (ii) our construction of a hybrid-scheme (cf. Theorem 5 in Section 5) and (iii) Theorem 2 above.

**Theorem 3** (Main theorem)**.** *Let $\mathcal{C}$ be a $(\mathsf{lb}_{\mathtt{code}}, q)$-CNMLR code. Then there exists an efficient RAM scheme $\mathsf{RS}$ and a constant $c = O(1)$ such that $\mathsf{RS}$ is RAM-simulatable in split-state tampering and $(\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathtt{bus}})$-leakage for $\mathsf{lb}_{\mathtt{disk}} + (c+1)\mathsf{lb}_{\mathtt{bus}} \leq \mathsf{lb}_{\mathtt{code}}$.*

# 4 Hybrid-to-Real Emulator

We construct an emulator $\mathcal{E}$ which for any constant $c$ efficiently transforms a $c$-bounding, extra-space insensitive hybrid scheme $\mathsf{HS} = (\mathbf{C}_\mathsf{h}, \mathbf{R}_\mathsf{h})$ into a RAM-scheme $\mathsf{RS} = (\mathbf{C}, \mathbf{R})$. Let $\mathcal{C} = (\text{Init}, \text{Encode}, \text{Decode})$ be a CNMLR code (cf. Section 2.2). If $\mathcal{C}$ tolerates leakage $\mathsf{lb}_{\mathtt{code}}$, then $\mathsf{RS}$ will tolerate any leakage $\mathsf{lb} = (\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathtt{bus}})$ with $\mathsf{lb}_{\mathtt{disk}} + (c+1)\mathsf{lb}_{\mathtt{bus}} \leq \mathsf{lb}_{\mathtt{code}}$. Moreover, $\mathsf{RS}$ will be secure against arbitrary tampering with the secret disks $\mathcal{SD}_1$ and $\mathcal{SD}_2$ and the public disk $\mathcal{PD}$.

**The Emulator $\mathcal{E}$.** Recall that the goal of the emulator $\mathcal{E}$ is to transform a hybrid RAM scheme $\mathsf{HS} = (\mathbf{C}_\mathsf{h}, \mathbf{R}_\mathsf{h})$ into a RAM-scheme $\mathsf{RS} = (\mathbf{C}, \mathbf{R})$. In particular, the emulator needs to specify transformations for the components of $\mathsf{HS}$. This includes the contents of the disks as well as the way instructions are stored and processed by $\mathsf{CPU}$. We provide an overview of the construction of the emulator; the details can be found in Fig. 3. For each location of the secret disk of $\mathbf{R}_\mathsf{h}$ it encodes the corresponding value with the CNMLR code. Each encoding consists of two halves, that are stored in the two secret disks $\mathcal{SD}_1$, $\mathcal{SD}_2$ of $\mathbf{R}$ (one half for each disk). Notice that the instructions that are stored on the public disk $\mathcal{PD}^\mathsf{h}$ in $\mathbf{R}_\mathsf{h}$ have to be adjusted in order to take care of the fact that $\mathbf{R}$ has two secret disks while $\mathbf{R}_\mathsf{h}$ only uses a single secret disk $\mathcal{SD}^\mathsf{h}_1$. This adjustment is straightforward: each time when an instruction asks the CPU to read from $\mathcal{SD}^\mathsf{h}_1$ in $\mathbf{R}_\mathsf{h}$, in $\mathbf{R}$ we read from the corresponding positions in $\mathcal{SD}_1$ and $\mathcal{SD}_2$. $\mathsf{CPU}$ in $\mathbf{R}$ will load the two parts of

an encoding from the secret disks and decode the codeword. If the codeword is invalid, i.e., the decoding outputs $\perp$, $\mathsf{CPU}$ self-destructs. Otherwise it will use the decoded value and use it as input for $\mathsf{CPU}^{\mathsf{h}}$.

**Theorem 4.** *If $\mathcal{C}$ is a $(\mathsf{lb}_{\mathtt{code}}, q)$-CNMLR code, then there exists a $(c, \mathsf{lb}, q)$-bounded hybrid-to-real emulator for any $\mathsf{lb} = (\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathtt{bus}})$ such that $\mathsf{lb}_{\mathtt{disk}} + (c+1)\mathsf{lb}_{\mathtt{bus}} \leq \mathsf{lb}_{\mathtt{code}}$.*

The proof of Theorem 4 can be found in Section 4.1; before coming to the proof, let us discuss some intuition. We need to show security of the emulator $\mathcal{E}$ according to Definition 11. To this end we prove that for any adversary $\mathcal{A}$ that runs in the real world $\text{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}$ and attacks the transformed RAM-scheme $\mathsf{RS} = \mathcal{E}(\mathsf{HS})$, there exists an adversary (simulator) $\mathcal{B}$ that runs in the hybrid world $\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$ and interacts with the hybrid-scheme $\mathsf{HS}$. The simulator $\mathcal{B}$ runs $\mathcal{A}$ as a sub-routine and simulates $\mathcal{A}$'s environment as in $\text{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}$. The main challenge is to simulate the `LEAK`/`TAMPER` queries (to split-state disks $\mathcal{SD}_1, \mathcal{SD}_2$ of $\mathbf{R}$) given only access to the `REPLACE` and `COPY` commands to the single secret disk $\mathcal{SD}^{\mathsf{h}}_1$.

The simulation works in two phases: the pre-processing (cf. Figure 4) and the online phase (cf. Figure 5). In the pre-processing phase the simulator $\mathcal{B}$ obtains the content $\omega^{\mathsf{h}}_0$ of the public disk $\mathcal{PD}^{\mathsf{h}}$ from its challenger and transforms it using the emulator $\mathcal{E}$. The main difficulty is to simulate the tampering and leakage access to the secret disks $\mathcal{SD}_1$ and $\mathcal{SD}_2$. Initially, in the pre-processing $\mathcal{B}$ creates encodings of 0 using the CNMLR code, and puts $(v_1, v_2) \leftarrow \text{Encode}(\Omega, 0)$ on the corresponding virtual/simulated disks. Depending on the queries in the online phase $\mathcal{B}$ will update these virtual disks in the following. `TAMPER` queries are simulated easily by applying the corresponding tamper functions to the current state of the virtual disks $\mathcal{SD}_1$ and $\mathcal{SD}_2$. Notice that also the leakage from the disks and the buses will essentially be done using the contents of the virtual disks. Hence, the main challenge of the simulation is how to keep these virtual disks consistent with what the adversary expects to see from an `EXEC` query. This is done by a rather involved case analysis and we only give the main idea here.

We distinguish the case when all the values on the secret disk that are used to evaluate the current instruction are *public* or at least some are *secret*. The first case may happen if the adversary $\mathcal{A}$ replaces the contents of the secret disks with some encoding of his choice by tampering. Notice that in this case the simulation is rather easy as $\mathcal{B}$ "knows" all the values and can simulate the execution of $\mathsf{CPU}$ (including the outputs and the new contents of the disks). If, on the other hand, some values that are used by $\mathsf{CPU}$ in the current execution are secret, then $\mathcal{B}$'s only chance to simulate $\mathcal{A}$ is to run $\mathsf{CPU}^{\mathsf{h}}$ on its hybrid challenger. The difficulty is to keep the state of the secret hybrid disk $\mathcal{SD}^{\mathsf{h}}_1$ consistent with the contents of the virtual disks $\mathcal{SD}_1, \mathcal{SD}_2$ maintained by $\mathcal{A}$. This is achieved by careful book-keeping and requires $\mathcal{B}$ to make use of his `REPLACE` and `COPY` commands to the single secret disk $\mathcal{SD}^{\mathsf{h}}_1$. The simulator $\mathcal{B}$ manages this book-keeping by using two records: (i) the set $S$ that stores encodings $(v_1, v_2)$ corresponding to values unknown to $\mathcal{B}$ (either generated during the pre-processing, or resulting from an evaluation of $\mathsf{CPU}^{\mathsf{h}}$ on partially secret inputs); (ii) the backup storage $\mathcal{BP}$ that $\mathcal{B}$ maintains on the hybrid secret disk $\mathcal{SD}^{\mathsf{h}}_1$ and stores a copy of all values that are unknown to the adversary (essentially, the values on $\mathcal{BP}$ correspond to the values that the encodings in $S$ are supposed to encode). Then the simulator can always copy the corresponding secret value to the position on $\mathcal{SD}^{\mathsf{h}}_1$ which corresponds to the value that *should* have been inside the encoding on the same position on the two secret virtual disk.

**Pre-processor:** The pre-processor $\mathbf{C} = \mathcal{E}(\mathbf{C}_{\mathsf{h}})$ runs as follows: Sample $(P, \omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}}) \leftarrow \mathbf{C}_{\mathsf{h}}(\mathcal{G}, \mathsf{K})$. Output $(P, \omega_0, \omega_1, \omega_2)$, where the computations of the public disk $\omega_0$ and the secret disks $\omega_1, \omega_2$ are detailed below.

**Secret Disks:** The secret disk of the hybrid pre-processing is encoded by encoding each memory position using the CNMC $\mathcal{C}$: Sample $\Omega \leftarrow \mathrm{Init}(1^k)$, and for $i = 0, \ldots, |\omega_1^{\mathsf{h}}| - 1$, let $v = \omega_1^{\mathsf{h}}[i]$, sample $(v_1, v_2) \leftarrow \mathrm{Encode}(\Omega, v)$ and set $\omega_1[i] = v_1$ and set $\omega_2[i] = v_2$.

**Public Disk:** Each instruction $(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$ from the public disk is compiled as follows:

- The instruction label is compiled as follows: $\mathcal{E}(\mathsf{Y}^{\mathsf{h}}) = \mathsf{Y}^{\mathsf{h}}$.
- A memory position $(\mathtt{src}_j^{\mathsf{h}}, \mathtt{loc}_j^{\mathsf{h}})$ is compiled as follows: If $\mathtt{src}_j^{\mathsf{h}} = 0$, then $\mathtt{src}_j = 0$, $\mathtt{loc}_j = \mathtt{loc}_j^{\mathsf{h}}$, and $\mathcal{E}(\mathtt{src}_j^{\mathsf{h}}, \mathtt{loc}_j^{\mathsf{h}}) = (\mathtt{src}_j, \mathtt{loc}_j)$. For notational convenience, assume that there are always $p$ such public positions and that they appear before the secret positions, this can be accomplished by adding dummy positions. Each secret position, i.e., with $\mathtt{src}_j^{\mathsf{h}} = 1$, compiles into two locations $(\mathtt{src}_{j'}, \mathtt{loc}_{j'}), (\mathtt{src}_{j'+1}, \mathtt{loc}_{j'+1})$ with $j' = 2j - p$, $\mathtt{src}_{j'} = 1$ and $\mathtt{src}_{j'+1} = 2$ and $\mathtt{loc}_{j'} = \mathtt{loc}_{j'+1} = \mathtt{loc}_j$. Let $\mathcal{E}(\mathtt{src}_j^{\mathsf{h}}, \mathtt{loc}_j^{\mathsf{h}}) = ((\mathtt{src}_{j'}, \mathtt{loc}_{j'}), (\mathtt{src}_{j'+1}, \mathtt{loc}_{j'+1}))$.
- Let $\mathsf{I}^{\mathsf{h}} = (I_1, \ldots, I_d)$ and $\mathcal{E}(\mathsf{I}^{\mathsf{h}}) = \mathcal{E}(I_1) \| \cdots \| \mathcal{E}(I_d)$, i.e., transform each element as above.
- Let $\mathsf{O}^{\mathsf{h}} = (O_1, \ldots, O_d)$ and $\mathcal{E}(\mathsf{O}^{\mathsf{h}}) = \mathcal{E}(O_1) \| \cdots \| \mathcal{E}(O_d)$.
- Let $\mathcal{E}(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}}) = (\mathcal{E}(\mathsf{Y}^{\mathsf{h}}), \mathcal{E}(\mathsf{I}^{\mathsf{h}}), \mathcal{E}(\mathsf{O}^{\mathsf{h}}))$.

The initial contents of the public disk $\omega_0^{\mathsf{h}} = (V_0, \ldots, V_{\ell-1})$ compiles as follows: let $\mathcal{E}(\omega_0^{\mathsf{h}}) = (\mathcal{E}(V_0), \ldots, \mathcal{E}(V_{\ell-1}))$, and let $\omega_0 = \mathcal{E}(\omega_0^{\mathsf{h}})$. We also define an inverse of the emulator. For an instruction of the compiled form $(\mathsf{Y}, \mathsf{I}, \mathsf{O}) = \mathcal{E}(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$, we let $\mathcal{E}^{-1}(\mathsf{Y}, \mathsf{I}, \mathsf{O}) = (\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$. For an input $X$ not of the compiled form, we let $\mathcal{E}^{-1}(X) = \mathtt{sd}$, where $\mathtt{sd}$ is some fixed input not of the form $(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$. Note that if and when the CPU of the hybrid scheme reads up $\mathtt{sd}$ it will self-destruct, see line 3 in Algorithm 1.

**CPU:** The compiled CPU $(\mathsf{Random}, \mathsf{Compute}) = \mathcal{E}(\mathsf{Random}^{\mathsf{h}}, \mathsf{Compute}^{\mathsf{h}})$ works as follows: $\mathsf{Random} = \mathsf{Random}^{\mathsf{h}}$ and $((\mathtt{O}_1, \ldots, \mathtt{O}_{2d-p}), \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{Compute}((\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_{2d-p}), r, \mathtt{pc}, \mathtt{ac})$ is specified by:

1. If $\mathcal{E}^{-1}(\mathtt{R}_0) \neq \mathtt{sd}$, then self destruct. Otherwise, compute $(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}}) = \mathcal{E}^{-1}(\mathtt{R}_0)$, and set $\mathtt{R}_0^{\mathsf{h}} = (\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$.
2. For $j = 1 \ldots, d$, let $(\mathtt{src}_j^{\mathsf{h}}, \mathtt{loc}_j^{\mathsf{h}}) = \mathsf{I}^{\mathsf{h}}[j]$. If $\mathtt{src}_j^{\mathsf{h}} = 0$, then let $\mathtt{R}_j^{\mathsf{h}} = \mathtt{R}_j$. If $\mathtt{src}_j^{\mathsf{h}} = 1$, then let $(v_1, v_2) = (\mathtt{R}_{j'}, \mathtt{R}_{j'+1})$, where $j' = 2j - p$. Let $v = \mathrm{Decode}(\Omega, (v_1, v_2))$. If $v = \perp$, then self destruct. Otherwise, let $\mathtt{R}_j^{\mathsf{h}} = v$.
3. Compute $((\mathtt{O}_1^{\mathsf{h}}, \ldots, \mathtt{O}_d^{\mathsf{h}}), \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{Compute}((\mathtt{R}_0^{\mathsf{h}}, \mathtt{R}_1^{\mathsf{h}}, \ldots, \mathtt{R}_d^{\mathsf{h}}), r, \mathtt{pc}, \mathtt{ac})$.
4. For $j = 1 \ldots, d$, let $(\mathtt{tar}_j, \mathtt{loc}_j) = \mathsf{O}^{\mathsf{h}}[j]$. If $\mathtt{tar}_j^{\mathsf{h}} = 0$, then set $\mathtt{O}_j \leftarrow \mathtt{O}_j^{\mathsf{h}}$. If $\mathtt{tar}_j^{\mathsf{h}} = 1$, then let $v = \mathtt{O}_j^{\mathsf{h}}$, sample $(v_1, v_2) \leftarrow \mathrm{Encode}(\Omega, v)$, and let $(\mathtt{O}_{j'}, \mathtt{O}_{j'+1}) \leftarrow (v_1, v_2)$, where $j' = 2j - p$.

**Dimensions:** The length of the disks will be the same. The word size on the public disks will be the same. The word size of the produced RAM will be large enough to hold an encoding under $\mathcal{C}$ as produced by the above code.

**Figure 3:** The Emulator, $\mathcal{E}$

## 4.1 Proof of Theorem 4

A formal description of the emulator $\mathcal{E}$ can be found in Fig. 3. The simulator $\mathcal{B}$ that we need to exhibit for proving Theorem 4 is depicted in Fig. 4 (pre-processing) and Fig. 5 (online). To

**Pre-processing:** The hybrid game will sample $(P, \omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}}) \leftarrow \mathbf{C}_{\mathsf{h}}(\mathcal{G}, \mathsf{K})$ and give $P$ to the simulator $\mathcal{B}$ and store $\omega_0^{\mathsf{h}}$ and $\omega_1^{\mathsf{h}}$ on $D_0^{\mathsf{h}}$ and $D_1^{\mathsf{h}}$ respectively. The simulator passes $P$ to $\mathcal{A}$. The simulator $\mathcal{B}$ can read all of $\omega_0^{\mathsf{h}}$ but nothing of $\omega_1^{\mathsf{h}}$ and has to create simulated disks $(D_0, D_1, D_2)$. We will identify $D_0$ with $\mathcal{E}(D_0^{\mathsf{h}}) = \mathcal{E}(\omega_0^{\mathsf{h}})$. Then create simulated disks $D_1$ and $D_2$ as follows: Let $L = |\omega_1^{\mathsf{h}}|$. Then sample $\Omega \leftarrow \mathrm{Init}(1^k)$. For $j = 0, \ldots, L-1$, sample $(v_{j,1}, v_{j,2}) \leftarrow \mathrm{Encode}(\Omega, 0)$, let $S[j] = (v_{j,1}, v_{j,2})$, $D_1[j] = v_{j,1}$ and $D_2[j] = v_{j,2}$, and then choose a backup location $\mathcal{BP}(j)$ on the secret disk such that $D_1^{\mathsf{h}}[\mathcal{BP}(j)]$ is never accessed by $\mathsf{HS}$ or $\mathcal{A}$,[a] and issue the command $(\texttt{COPY}, j, \mathcal{BP}(j))$ to create a back up of the value $D_1^{\mathsf{h}}[j] = \omega_1^{\mathsf{h}}[j]$. Notice that in this way the simulator $\mathcal{B}$ keeps a copy of the original secret value that $(v_{j,1}, v_{j,2})$ is supposed to encode (instead of 0 as after the pre-processing of $\mathcal{B}$). Finally, let $\mathsf{ns} = L$.

[a] We can do this as we assume the hybrid scheme is extra-space insensitive, so we can expand the secret disk beyond with index space used by $\mathsf{HS}$ and $\mathcal{A}$ and use this extra space for back-up.

**Figure 4:** The Simulator, $\mathcal{B}^{\mathcal{A}}$, Pre-Processing

conclude the proof we need to show that the view produced by the hybrid simulator $\mathcal{B}$ (interacting with $\mathcal{A}$ in $\mathrm{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}$) is computationally indistinguishable from the view that $\mathcal{A}$ obtains in the real experiment $\mathrm{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)$. We do so via a reduction to the adaptive composability property of the CNMLR code $\mathcal{C}$ (cf. Definition 2).

**Reduction to the CNMLR code.** The reduction $\mathcal{R}$ (depicted in Fig. 6-7) has access to the leakage oracles $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1, \cdot), \mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_2, \cdot)$ and tamper oracle $\mathcal{O}^q_{\mathsf{comp}}((\mathbf{c}_1, \mathbf{c}_2), (\cdot, \cdot))$ from $\mathrm{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{R}}(b)$ (for a random $b \in \{0, 1\}$). The main difficulty is to make sure that $\mathcal{R}$ indeed can virtually run the hybrid simulator $\mathcal{B}$ in a way that is consistent with the encodings that are produced inside the target oracles $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1, \cdot), \mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_2, \cdot)$ and $\mathcal{O}^q_{\mathsf{comp}}((\mathbf{c}_1, \mathbf{c}_2), (\cdot, \cdot))$. To this end, $\mathcal{R}$ first runs the hybrid compiler to obtain $(P, \omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}}) \leftarrow \mathbf{C}_{\mathsf{h}}(\mathcal{G}, \mathsf{K})$. During the following execution of the game, the reduction $\mathcal{R}$ ensures that what is stored inside its challenge oracles $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1, \cdot), \mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_2, \cdot), \mathcal{O}^q_{\mathsf{comp}}((\mathbf{c}_1, \mathbf{c}_2), (\cdot, \cdot))$ can be kept consistent with the contents on the hybrid secret disk $\omega_1^{\mathsf{h}}$ (and hence with the simulated virtual disks $\mathcal{SD}_1, \mathcal{SD}_2$). $\mathcal{R}$ uses so-called *disk reconstruction functions* $\mathsf{Dcon}_i$ that take as input a set of encodings $\mathbf{c}_i$ (this is the current state of the target oracles) and reconstructs the content of the corresponding secret disks $\mathcal{SD}_i$. Given such functions $\mathsf{Dcon}_i$, simulating the `TAMPER` and `LEAK` queries can be easily done by concatenating the tamper and leakage functions submitted by $\mathcal{A}$ with the current disk reconstruction functions $\mathsf{Dcon}_i$. One main tedious difficulty in the reduction is to continuously update the disk reconstruction functions such that they are consistent with what the adversary $\mathcal{A}$ expects to see. For instance, if $\mathcal{A}$ asks for a `TAMPER` query $(\mathsf{Tamper}(.), i)$ then $\mathsf{Dcon}_i$ is updated by concatenating $\mathsf{Tamper}$ with the current $\mathsf{Dcon}_i$, i.e., we get $\mathsf{Dcon}'_i = \mathsf{Dcon}_i \circ \mathsf{Tamper}$. The full details about how $\mathcal{R}$ maintains $\mathsf{Dcon}_i$ are given in Fig. 6-7.

One can verify that if in the simulation we initialize the secret disks $\mathcal{SD}_1, \mathcal{SD}_2$ with encodings of the correct secret values, then the simulator $\mathcal{B}$ from Fig. 4–5 produces exactly the distribution as in $\mathrm{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}$. Hence, the reduction will essentially run the code of $\mathcal{B}$ and submits to its target oracles inputs of the form $(0, \omega_1^{\mathsf{h}}[j])$ in each iteration of the loop. Depending on the challenge bit $b$, the reduction either simulates $\mathrm{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}$ (if $b = 0$) or $\mathrm{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}$ (if $b = 1$), i.e., we have:

$$\mathrm{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{R}}(0) \equiv \mathrm{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}} \qquad \text{and} \qquad \mathrm{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{R}}(1) \equiv \mathrm{TLREAL}^{\mathsf{lb},q}_{\mathsf{RS},\mathcal{A},\mathcal{G}}.$$

**Online:** Given a command $\texttt{CMD}$ from $\mathcal{A}$ the simulator $\mathcal{B}$ acts as follows:

1. If $\texttt{CMD} = (\texttt{STOP}, \mathsf{O}_{\texttt{real}})$ then issue command $(\texttt{STOP}, \mathsf{O}_{\texttt{real}})$ and halt.
2. If $\texttt{CMD} = (\texttt{LEAK}, i, \mathsf{Leak}(\cdot))$, proceed as follows. If $i = 0$, compute $\lambda \leftarrow \mathsf{Leak}(D_i)$ and give $\lambda$ to $\mathcal{A}$. Otherwise, update the total leakage, $\texttt{lt}_i \leftarrow \texttt{lt}_i + |\mathsf{Leak}|$, and only if $\texttt{lt}_i \leq \mathsf{lb}_{\texttt{disk}}$ then compute $\lambda \leftarrow \mathsf{Leak}(D_i)$ using the secret disks and give $\lambda$ to $\mathcal{A}$.
3. If $\texttt{CMD} = (\texttt{TAMPER}, i, \mathsf{Tamper}(\cdot))$ then proceed as follows: If $i = 0$, then let $D_0 \leftarrow \mathsf{Tamper}(D_0)$ and use $\texttt{REPLACE}$ commands to let $D_0^{\mathsf{h}} = \mathcal{E}^{-1}(D_0)$. If $i > 0$, let $D_i \leftarrow \mathsf{Tamper}(D_i)$ for the simulated disk $D_i$.
4. If $\texttt{CMD} = (\texttt{EXEC}, \mathsf{Leak}_1, \mathsf{Leak}_2)$ and $\texttt{B} = 0$ then do the following:
   (a) If $\mathcal{E}^{-1}(D_0[\texttt{pc}]) \neq \texttt{sd}$, then self destruct. Otherwise, compute $(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}}) = \mathcal{E}^{-1}(D_0[\texttt{pc}])$, and set $\mathsf{R}_0^{\mathsf{h}} = (\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$.
   (b) For $j = 1 \ldots, d$, let $(\texttt{src}_j^{\mathsf{h}}, \texttt{loc}_j^{\mathsf{h}}) = \mathsf{I}^{\mathsf{h}}[j]$. If $\texttt{src}_j^{\mathsf{h}} = 0$, then let $\mathsf{R}_j^{\mathsf{h}} = D_0[\texttt{loc}_j]$. If $\texttt{src}_j^{\mathsf{h}} = 1$, then let $(v_1, v_2) = (D_1[\texttt{loc}_j^{\mathsf{h}}], D_2[\texttt{loc}_j^{\mathsf{h}}])$. If $\exists g : (v_1, v_2) = S[g]$, then issue the command $(\texttt{COPY}, \mathcal{BP}(g), \texttt{loc}_j^{\mathsf{h}})$ to put back in $D_1^{\mathsf{h}}[\texttt{loc}_j^{\mathsf{h}}]$ the value that $(v_1, v_2)$ *should* have been an encoding of. If $\nexists g : (v_1, v_2) = S[g]$, then compute $v = \mathrm{Decode}_\Omega(v_1, v_2)$. If $v = \perp$, then simulate a self destruct (by ignoring all future $\texttt{EXEC}$ commands). Otherwise, issue the command $(\texttt{REPLACE}, 1, \texttt{loc}_j^{\mathsf{h}}, v)$ to put in $D_1^{\mathsf{h}}[\texttt{loc}_j^{\mathsf{h}}]$ the value that $(v_1, v_2)$ is an encoding of.
   (c) Issue the command $\texttt{CMD} = (\texttt{EXEC})$ to run $\mathsf{CPU}^{\mathsf{h}}$, which replaces the existing disks by the modified output: $(D_0^{\mathsf{h}}, D_1^{\mathsf{h}}, \texttt{B}, \texttt{T}) \leftarrow \mathsf{CPU}^{\mathsf{h}}(\texttt{pc}, \texttt{ac}, D_0^{\mathsf{h}}, D_1^{\mathsf{h}})$. This also updates $D_0$ as we identify $D_0 = \mathcal{E}(D_0^{\mathsf{h}})$.
   (d) Then update the simulated disks $D_1$ and $D_2$. For $j = 1 \ldots, d$, let $(\texttt{src}_j^{\mathsf{h}}, \texttt{loc}_j^{\mathsf{h}}) = \mathsf{O}^{\mathsf{h}}[j]$. How we process each $(\texttt{src}_j^{\mathsf{h}}, \texttt{loc}_j^{\mathsf{h}})$ depends on whether the above execution was a *secret execution*, in the sense of the case $\texttt{CMD} = (\texttt{EXEC})$ in Fig. 2.

   *public*: If $\texttt{src}_j^{\mathsf{h}} = 1$, then let $v = D_i^{\mathsf{h}}[\texttt{loc}_j^{\mathsf{h}}]$,[a] sample $(v_1, v_2) \leftarrow \mathrm{Encode}(\Omega, v)$ and let $(D_1[\texttt{loc}_j^{\mathsf{h}}], D_2[\texttt{loc}_j^{\mathsf{h}}]) \leftarrow (v_1, v_2)$.

   *secret*: If $\texttt{src}_j^{\mathsf{h}} = 1$, then sample $(v_1, v_2) \leftarrow \mathrm{Encode}(\Omega, 0)$ and let $(D_1[\texttt{loc}_j^{\mathsf{h}}], D_1[\texttt{loc}_j^{\mathsf{h}}]) \leftarrow (v_1, v_2)$. Then let $S[\mathsf{ns}] \leftarrow (v_1, v_2)$, pick a fresh back-up location $\mathcal{BP}(\mathsf{ns})$ and issue the command $(\texttt{COPY}, \texttt{loc}_j^{\mathsf{h}}, \mathcal{BP}(\mathsf{ns}))$ to back up the value that $(v_1, v_2)$ *should* have been an encoding of, and then let $\mathsf{ns} \leftarrow \mathsf{ns} + 1$.

   (e) Finally simulate the leakage $\lambda_1 = \mathsf{Leak}_1(\mathsf{Bs}_1)$ and $\lambda_2 = \mathsf{Leak}_2(\mathsf{Bs}_2)$ by computing $\mathsf{Bs}_1$ and $\mathsf{Bs}_2$ as in the real world, but using the above simulated values. In particular, $\mathsf{Bs}_i \ni v_i$ for all $(v_1, v_2) = (D_1[\texttt{loc}_j^{\mathsf{h}}], D_2[\texttt{loc}_j^{\mathsf{h}}])$ from the reading and all $(v_1, v_2) \leftarrow \mathrm{Encode}(\Omega, 0)$ from the writing.

[a] When the execution is public, then $\mathcal{B}$ knows all inputs to the CPU and hence can compute all the outputs and hence $D_i^{\mathsf{h}}[\texttt{loc}_j]$. This is not completely true, as the CPU could be randomized. However, in that case $\mathcal{B}$ could first run the CPU by issuing the command $(\texttt{EXEC})$. Then it could internally run $\mathsf{Random}$ and $\mathsf{Compute}$ to recompute the CPU on the same inputs, but with fresh randomness. Then it can use $\texttt{REPLACE}$ commands to write the resulting outputs to their respective locations. Since the CPU cannot keep any information about the randomness used in previous executions, this simulation will result in exactly the same distribution, and now $\mathcal{B}$ knows the values it needs.

**Figure 5:** The Simulator, $\mathcal{B}^{\mathcal{A}}$, Online

Security of the emulator $\mathcal{E}$ now follows from the adaptive composability of the CNMLR code.

**Pre-processing:** Sample $(P, \omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}}) \leftarrow \mathbf{C}_{\mathsf{h}}(\mathcal{G}, \mathsf{K})$ and give $P$ to $\mathcal{A}$. Let $D_0 = \mathcal{E}(\omega_0^{\mathsf{h}})$ and $D_1^{\mathsf{h}} = \omega_1^{\mathsf{h}}$. Then create virtual disks $D_1 = \omega_1$ and $D_2 = \omega_2$ "inside the leakage oracles" $\mathcal{O}^{\mathsf{lb}_{\mathtt{code}}}(\mathbf{c}_1, \cdot)$ and $\mathcal{O}^{\mathsf{lb}_{\mathtt{code}}}(\mathbf{c}_2, \cdot)$ by maintaining *disk reconstruction functions* $\mathsf{Dcon}_1$, $\mathsf{Dcon}_2$ such that $D_i = \mathsf{Dcon}_i(\mathbf{c}_i)$. Initially $\mathsf{Dcon}_i$ is the function outputting 0 on all inputs $j$. We elaborate on how to maintain $\mathsf{Dcon}_i$ below. Let $L = |\omega_1^{\mathsf{h}}|$. For $j = 0, \ldots, L-1$, output $(0, \omega_1^{\mathsf{h}}[j])$ to the game $\textsc{Game}_{\mathcal{C},\mathcal{R}}^{\mathsf{comp},q,\mathsf{lb}_{\mathtt{code}}}(b)$ to make it create an encoding $(v_{j,1}, v_{j,2})$ of either 0 or $\omega_1^{\mathsf{h}}[j]$ and add $v_{j,1}$ to $\mathbf{c}_1[j]$ $v_{j,2}$ to $\mathbf{c}_2[j]$. Notice that the record $S$ kept by $\mathcal{B}$ in Figure 4 is now represented by $(\mathbf{c}_1, \mathbf{c}_2)$: The reduction maintains the invariant that whenever $\mathcal{B}$ would have sampled $(v_1, v_2)$ and stored it in $S[g]$, the reduction makes an encoding query to its challenge oracle as specified in Definition 2, and this will be the $g$-th query, such that $(\mathbf{c}_1[g], \mathbf{c}_2[g]) = (v_1, v_2)$. Update the disk reconstruction functions $\mathsf{Dcon}_1$, $\mathsf{Dcon}_2$ as follows: Let $\mathsf{Dcon}_i$ be the function before the $L$ encodings were made. Then let $\mathsf{Dcon}_i' = \mathsf{Dcon}_i$, except that $(\mathsf{Dcon}_i'(\mathbf{c}_i))[j] = \mathbf{c}_i[j]$ for $j = 0, \ldots, L-1$. Furthermore, for each $j$, let $\mathcal{BP}(j)$ be the back-up location chosen by $\mathcal{B}$ and simulate the commands $(\mathtt{COPY}, j, \mathcal{BP}(j))$ by setting $D_1^{\mathsf{h}}[\mathcal{BP}(j)] = D_1^{\mathsf{h}}[j]$.

**Figure 6:** The Reduction, $\mathcal{R}^{\mathcal{A}}$

**Computing the leakage bound.** We finally argue about why our reduction satisfies the leakage bounds of the target oracles. First, observe that since $\mathsf{HS}$ is $c$-bounding we have that except with negligible probability the simulator $\mathcal{B}$ would access each value at most $c$ times. By construction, this means that each encoding in the reduction will be part of at most $c$ leakage queries to simulate the execution of the bus—we elaborate on this claim below. Each such leakage query leaks at most $\mathsf{lb}_{\mathtt{bus}}$ bits, for a total of $c \cdot \mathsf{lb}_{\mathtt{bus}}$. Furthermore, each encoding might enter into the leakage queries to simulate leakage form the disk, but at most $\mathsf{lb}_{\mathtt{disk}}$ bits are needed for this. Finally, in the activation where the self-destruct happens the reduction might request further $\mathsf{lb}_{\mathtt{bus}}$ bits of leakage. Hence, except with negligible probability the reduction $\mathcal{R}$ requests at most $(c+1)\mathsf{lb}_{\mathtt{bus}} + \mathsf{lb}_{\mathtt{disk}}$ bits of leakage from each encoding. Then use that $(c+1)\mathsf{lb}_{\mathtt{bus}} + \mathsf{lb}_{\mathtt{disk}} \leq \mathsf{lb}_{\mathtt{code}}$, where $\mathsf{lb}_{\mathtt{code}}$ is the leakage tolerated by the CNMLR code $\mathcal{C}$.

Now, let us explain why $\mathsf{HS}$ being $c$-bounding implies that except with negligible probability the simulator $\mathcal{B}$ will access each value in $\mathbf{c}_i$ at most $c$ times when simulating the $\mathtt{EXEC}$ command. Notice that the query to the leakage oracle occurs in Step 4e in Figure 7 and only if the execution was secret. Here the leakage function $\mathsf{Leak}_i'$ needs to compute $\mathsf{Bs}_i$. The value $\mathsf{Bs}_i$ contains values from the bus of the reading phase and the writing phase.

Let us start by discussing the writing phase, as this is the easier case. Here $\mathsf{Leak}_i'$ computes the disk $D_i' = \mathsf{Dcon}_i'(\mathbf{c}_i)$ as it looked after the writing phase, and adds to $\mathsf{Bs}_i$ each $v_i = D_i[\mathtt{loc}_j^{\mathsf{h}}]$ from the writing phase. Notice that, however, for each $D_i[\mathtt{loc}_j^{\mathsf{h}}]$ there exists an index $g$ defined in Step 4d such that $D_i[\mathtt{loc}_j^{\mathsf{h}}] = \mathbf{c}_i[g]$ and this $g$ can clearly be computed by $\mathcal{R}$. Hence $\mathcal{R}$ can compute $D_i' = \mathsf{Dcon}_i'(\mathbf{c}_i)$: The leakage function simply adds each new $\mathbf{c}_i[g]$ from Step 4d to $\mathsf{Bs}_i$. This brings the leakage tally for $\mathbf{c}_i[g]$ up to at most $\mathsf{lb}_{\mathtt{bus}}$, as it is a fresh encoding and hence was not accessed before. Note that in the hybrid game the counter $C[g]$ is set to 1. In particular, the leakage tally of $\mathbf{c}_i[g]$ is less than $C[g] \cdot \mathsf{lb}_{\mathtt{bus}}$.

As for the the writing phase, the leakage function first computes the disk $D_i = \mathsf{Dcon}_i(\mathbf{c}_i)$ as it looked at the reading phase and then adds to $\mathsf{Bs}_i$ each value $v_i = D_i[\mathtt{loc}_j^{\mathsf{h}}]$. Note, however, that some of these values were computed by $\mathcal{R}$ already in Step 4b. Namely, the reduction made the tampering query $(\mathsf{T}_1, \mathsf{T}_2)$, where $\mathsf{T}_i(\mathbf{c}_i) = \mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}_j^{\mathsf{h}}]$. The reply is either $(\mathsf{same}^\star, g)$, $\perp$ or a

**Online:** Get command $\texttt{CMD}$ from $\mathcal{A}$ and act as follows according to the command-type:

1. If $\texttt{CMD} = (\texttt{STOP}, \mathsf{O}_{\texttt{real}})$ then output $\mathsf{O}_{\texttt{real}}$ and halt.
2. If $\texttt{CMD} = (\texttt{LEAK}, i, \mathsf{Leak}(\cdot))$, then: If $i = 0$, compute $\lambda \leftarrow \mathsf{Leak}(D_i)$ and return $\lambda$ to $\mathcal{A}$. If $i > 0$, then update the total leakage, $\texttt{lt}_i \leftarrow \texttt{lt}_i + |\mathsf{Leak}|$, and if $\texttt{lt}_i \le \mathsf{lb}_{\texttt{disk}}$ then compute $\lambda$ by submitting to $\mathcal{O}^{\mathsf{lb}_{\texttt{code}}}(\mathbf{c}_i, \cdot)$ the function $\mathsf{Leak}' = \mathsf{Leak} \circ \mathsf{Dcon}_i$ and return $\lambda = \mathcal{O}^{\mathsf{lb}_{\texttt{code}}}(\mathbf{c}_i, \mathsf{Leak}')$ to $\mathcal{A}$.
3. If $\texttt{CMD} = (\texttt{TAMPER}, i, \mathsf{Tamper}(\cdot))$ and $i = 0$, let $D_0 \leftarrow \mathsf{Tamper}(D_0)$ and $D_0^{\mathsf{h}} \leftarrow \mathcal{E}^{-1}(D_0)$. Otherwise, virtually modify the disk $D_i \leftarrow \mathsf{Tamper}(D_i)$ by modifying the function $\mathsf{Dcon}_i$ as follows: $\mathsf{Dcon}_i \leftarrow \mathsf{Tamper} \circ \mathsf{Dcon}_i$.
4. If $\texttt{CMD} = (\texttt{EXEC}, \mathsf{Leak}_1, \mathsf{Leak}_2)$ and $\texttt{B} = 0$ then do the following:
   (a) Read the instruction $(\mathsf{Y}, \mathsf{I}, \mathsf{O}) = D_0[\texttt{pc}]$. If the instruction $(\mathsf{Y}, \mathsf{I}, \mathsf{O})$ is not of the compiled form, then self destruct. Otherwise, compute $(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$ such that $(\mathsf{Y}, \mathsf{I}, \mathsf{O}) = \mathcal{E}(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$, and set $\texttt{R}_0^{\mathsf{h}} = (\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$.
   (b) For $j = 1 \ldots, d$, let $(\texttt{src}_j^{\mathsf{h}}, \texttt{loc}_j^{\mathsf{h}}) = \mathsf{I}^{\mathsf{h}}[j]$. If $\texttt{src}_j^{\mathsf{h}} = 0$, then let $\texttt{R}_j^{\mathsf{h}} = D_0[\texttt{loc}_j^{\mathsf{h}}]$. If $\texttt{src}_j^{\mathsf{h}} = 1$, then submit the tampering query $(\mathsf{T}_1, \mathsf{T}_2)$, where $\mathsf{T}_i(\mathbf{c}_i) = \mathsf{Dcon}_i(\mathbf{c}_i)[\texttt{loc}_j^{\mathsf{h}}]$. If the reply is $(\mathsf{same}^\star, g)$, simulate the command $(\texttt{COPY}, \mathcal{BP}(g), \texttt{loc}_j^{\mathsf{h}})$ by setting $D_1^{\mathsf{h}}[\texttt{loc}_j^{\mathsf{h}}] \leftarrow D_1^{\mathsf{h}}[\mathcal{BP}(g)]$. If the reply is $\perp$, then simulate a self destruct. Otherwise, compute $v = \mathrm{Decode}_\Omega(v_1, v_2)$ and simulate the command $(\texttt{REPLACE}, 1, \texttt{loc}_j^{\mathsf{h}}, v)$ by setting $D_1^{\mathsf{h}}[\texttt{loc}_j^{\mathsf{h}}] \leftarrow v$.
   (c) Compute $(D_0^{\mathsf{h}}, D_1^{\mathsf{h}}, \texttt{B}, \texttt{T}) \leftarrow \mathsf{CPU}^{\mathsf{h}}(\texttt{pc}, \texttt{ac}, D_0^{\mathsf{h}}, D_1^{\mathsf{h}})$. Update $\texttt{pc}$ and $\texttt{ac}$ and if the self destruct flag is set, simulate a self destruct. Let $D_0 = \mathcal{E}(D_0^{\mathsf{h}})$.
   (d) For $j = 1 \ldots, d$, let $(\texttt{src}_j^{\mathsf{h}}, \texttt{loc}_j^{\mathsf{h}}) = \mathsf{O}^{\mathsf{h}}[j]$. How we process each $(\texttt{src}_j^{\mathsf{h}}, \texttt{loc}_j^{\mathsf{h}})$ depends on whether the above execution was a *secret execution*.
   *public* If $\texttt{src}_j^{\mathsf{h}} = 1$, then let $v = D_i^{\mathsf{h}}[\texttt{loc}_j^{\mathsf{h}}]$, sample $(v_1, v_2) \leftarrow \mathrm{Encode}(\Omega, v)$ and update $\mathsf{Dcon}_i$ to $\mathsf{Dcon}_i' = \mathsf{Dcon}_i$, except that $(\mathsf{Dcon}_i'(\mathbf{c}_i))[\texttt{loc}_j^{\mathsf{h}}] = v_i$.
   *secret* If $\texttt{src}_j^{\mathsf{h}} = 1$, then issue the encoding request $(0, D_1^{\mathsf{h}}[\texttt{loc}_j^{\mathsf{h}}])$ to make $\textsc{Game}_{\mathcal{C},\mathcal{R}}^{\mathsf{comp},g,\mathsf{lb}_{\texttt{code}}}(b)$ generate an encoding $(v_{g,1}, v_{g,2})$ – assume this was the $g$'th encoding request. As a result $v_{g,i}$ is added to $\mathbf{c}_i$ as $\mathbf{c}_i[g]$. Define $\mathsf{Dcon}_i'$ = $\mathsf{Dcon}_i$ except that $(\mathsf{Dcon}_i'(\mathbf{c}_i))[\texttt{loc}_j^{\mathsf{h}}] = \mathbf{c}_i[g]$. Let $\mathcal{BP}(g)$ be the backup location used by $\mathcal{B}$ and simulate the command $(\texttt{COPY}, \texttt{loc}_j^{\mathsf{h}}, \mathcal{BP}(g))$ by setting $D_1^{\mathsf{h}}[\mathcal{BP}(g)] \leftarrow D_1^{\mathsf{h}}[\texttt{loc}_j^{\mathsf{h}}]$.
   (e) Finally simulate the leakage $\lambda_1 = \mathsf{Leak}_1(\mathsf{Bs}_1)$ and $\lambda_2 = \mathsf{Leak}_2(\mathsf{Bs}_2)$ by computing $\mathsf{Bs}_1$ and $\mathsf{Bs}_2$ as in the real world. If the execution was public the reduction knows all the needed values to do this. If the execution was secret, it will be done inside the leakage oracles. In particular, for $i = 1, 2$, submit to $\mathcal{O}^{\mathsf{lb}_{\texttt{code}}}(\mathbf{c}_i, \cdot)$ the function $\mathsf{Leak}_i'$ which first computes the disk $D_i = \mathsf{Dcon}_i(\mathbf{c}_i)$ as it looked at the reading phase and and then adds to $\mathsf{Bs}_i$ each value $v_i = D_i[\texttt{loc}_j^{\mathsf{h}}]$. Then it computes the disk $D_i' = \mathsf{Dcon}_i'(\mathbf{c}_i)$ as it looked after the writing phase and adds each $v_i = D_i[\texttt{loc}_j^{\mathsf{h}}]$ from the writing phase. Then it returns $\lambda_i = \mathsf{Leak}_i(\mathsf{Bs}_i)$.

**Figure 7:** The Reduction, $\mathcal{R}^{\mathcal{A}}$, Online

valid encoding. We look at each case separately:

- If the reply was not $(\mathsf{same}^\star, g)$ or $\perp$, then the reply from the tampering oracle was exactly $(v_1, v_2) = (\mathsf{Dcon}_1(\mathbf{c}_1)[\texttt{loc}_j^{\mathsf{h}}], \mathsf{Dcon}_2(\mathbf{c}_2)[\texttt{loc}_j^{\mathsf{h}}])$. Hence the reduction can hard-code the value

$v_i$ into the leakage query $\mathsf{Leak}_i$ and add it to $\mathsf{Bs}_i$ as the value $D_i[\mathtt{loc}^{\mathsf{h}}_j] = v_i$.

- If the reply was $(\mathsf{same}^\star, g)$, then since $\mathsf{T}_i(\mathbf{c}_i) = \mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}^{\mathsf{h}}_j]$, we have that $(D_1[\mathtt{loc}^{\mathsf{h}}_j], D_2[\mathtt{loc}^{\mathsf{h}}_j])$ is one of the encodings created by the game by request of the reduction, and the reduction knows which one, namely the $g$'th one.[14] So, the leakage function can compute $\mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}^{\mathsf{h}}_j]$ as $\mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}^{\mathsf{h}}_j] = \mathbf{c}_i[g]$, and here only these encodings $\mathbf{c}_i[g]$ are accessed. Hence each of them has their leakage tally increased by at most $\mathsf{lb}_{\mathtt{bus}}$. Notice that when the tampering returns $(\mathsf{same}^\star, g)$, then the reduction simulates the command $(\mathtt{COPY}, \mathcal{BP}(g), \mathtt{loc}^{\mathsf{h}}_j)$ by setting $D^{\mathsf{h}}_1[\mathtt{loc}^{\mathsf{h}}_j] \leftarrow D^{\mathsf{h}}_1[\mathcal{BP}(g)]$, which results in the value that $(\mathbf{c}_1[g], \mathbf{c}_2[g])$ should have been an encoding of to be placed in $D^{\mathsf{h}}_1[\mathtt{loc}^{\mathsf{h}}_j]$ and hence read up by the CPU in Step 4c. In the hybrid game, this would result in the counter $C[g]$ for that value to be incremented by one. Hence the leakage tally for each element stays below $C[g] \cdot \mathsf{lb}_{\mathtt{bus}} \leq c \cdot \mathsf{lb}_{\mathtt{bus}}$.
- If the reply was $\perp$, then there is no way around computing $D_i = \mathsf{Dcon}_i(\mathbf{c}_i)$ (and then computing $\mathsf{Bs}_i$ from $D_i$). I.e., the leakage function in the worst case accessed all encodings, as it might need to know the entire $\mathbf{c}_i$ to compute $D_i = \mathsf{Dcon}_i(\mathbf{c}_i)$. This can, however, happen at most once as the CPU can self-destruct at most once. This one extra "full disk" possible leakage of at most $\mathsf{lb}_{\mathtt{bus}}$ bits is why we get the bound $\mathsf{lb}_{\mathtt{disk}} + (c+1)\mathsf{lb}_{\mathtt{bus}}$ as opposed to $\mathsf{lb}_{\mathtt{disk}} + c\mathsf{lb}_{\mathtt{bus}}$.

# 5 The Hybrid Scheme

In this section we describe an $O(1)$-bounding, extra-space insensitive hybrid scheme $\mathsf{HS}$. Recall that the hybrid schemes $\mathsf{HS}$ consists of a hybrid ram $\mathbf{R}_{\mathsf{h}}$ and a hybrid compiler $\mathbf{C}_{\mathsf{h}}$ which takes a functionality $\mathcal{G}$ with secret key $\mathsf{K}$ and outputs an encoding of the form $(P, \omega^{\mathsf{h}}_0, \omega^{\mathsf{h}}_1)$ to be placed on the disks of the RAM. The RAM $\mathbf{R}_{\mathsf{h}}$ consists of a CPU $\mathsf{CPU}^{\mathsf{h}}$, which is specified by two functions $\mathsf{Random}$ and $\mathsf{Compute}$. Below, we present an outline of our hybrid scheme $\mathsf{HS}$ and refer the reader to the following subsections for the details.

**Overview of $\mathsf{HS}$.** We start by assuming that we have a "regular program" (i.e., a sequence of instructions) for computing $\mathcal{G}_{\mathsf{K}}$ in a "regular" RAM (i.e., a RAM with a public disk and a CPU without any security). This regular program essentially "encodes" the original functionality in a format that is compatible with the underlying RAM; for example the key is parsed as a sequence of words that are written in the corresponding locations of the public disk. The RAM needs to be neither tamper nor leakage resilient, and the "regularity" essentially comes from the fact that it emulates $\mathcal{G}_{\mathsf{K}}$ correctly and has no pathological behaviour, like overwriting the key during an activation. We also need that it reads each value $O(1)$ times. It easy to see that one can always translate the functionality into such a regular program, generically, using, e.g., a bounded fan-out circuit layed out as a RAM program. We refer the reader to Section 5.1 for the complete specifications.

Now, given such a regular program, our hybrid compiler $\mathbf{C}_{\mathsf{h}}$ is supposed to produce a *compiled* program (during the pre-processing phase) to be run by the hybrid RAM $\mathbf{R}_{\mathsf{h}}$ (during the on-line phase). The compiled program is placed on the public disk from which $\mathsf{CPU}^{\mathsf{h}}$ reads in sequence. Our CPU $\mathsf{CPU}^{\mathsf{h}}$ will be deterministic, and hence $\mathsf{Random}$ just outputs the empty string at each

[14] Here we use the property of composable CNMLR codes which ensures that the tampering oracle returns not only the symbol $\mathsf{same}^\star$ but also the index with which the tampered value matches.

invocation. This means that we only have to specify the compiler $\mathbf{C}_{\mathsf{h}}$ and the function $\mathsf{Compute}$ for a complete specification of $\mathsf{HS}$.

**Our hybrid scheme.** Recall that the adversary in a hybrid execution is only allowed a limited form of tampering, by which he can copy values within the secret disk and replace some value with a known one. The main idea will be to store the regular program (and all intermediary values) in the secret disk; each value will be stored in a special "augmented" form. The augmentation includes: (a) A secret label $L$ (sampled once and for all at setup, and thus unknown to the adversary); (b) The position $j$ at which the value is stored; (c) The current values $(a, p)$ of the activation and program counters $(\mathsf{ac}, \mathsf{pc})$ when the value was written. Intuitively, the secret label ensures that the adversary cannot use the "replace" command as that would require to guess the value of the label. On the other hand the position $j$ will allow the CPU to check that it loaded a value from the right position, preventing the adversary to use the "copy" command to move values created by the CPU (or at setup) to another location. Finally, the pair $(a, p)$ prevents the adversary from swapping values sharing the same $L$ and the same $j$ (i.e., *resetting* by forcing the CPU to re-use a previously encoded value).

Whenever algorithm $\mathsf{Compute}$ of the CPU loads some instruction, it uses the above augmented encodings to check that it is loading the right instruction, that the correct location was read, that the label matches, and that the counters are consistent; if any of the above fails, it self-destructs. Otherwise, it runs the specific instruction of the emulated regular program, and writes the resulting value to the disk (in the augmented form). A detailed description can be found in Section 5.2 (see Fig. 8–12). A complete security analysis is given in Section 5.3.

### 5.1 A Regular Program for $\mathcal{G}$

For simplicity we will assume to have a "regular" program for computing $\mathcal{G}_{\mathsf{K}}$ through a "regular" RAM, i.e., a random access machine with *one* public disk, *one* CPU and *no* secret disk. Such a regular RAM is not assumed to be neither leakage nor tamper resilient and, as we argue below, it can be assumed generically. All we require is that it computes $\mathcal{G}_{\mathsf{K}}$. In Section 5.2 we will compile this regular program into a hybrid-scheme (which can in turn be transformed into a RAM-scheme via the emulator of Section 4).

Suppose the RAM has word size $w$. In the description below the term *size* refers to the number of words, and the term *position* refers to the location of a word in the RAM. A program for a regular RAM with word size $w$ is specified by $(\ell_R, K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O, \mathcal{K}, \mathcal{X}, \mathcal{X}^{-1}, \mathcal{Y}, \mathcal{Y}^{-1}, g, (\iota_0, \ldots, \iota_{\ell_G-1}))$, where $\ell_R$ is the size of the RAM, $K$ is the position in the RAM where the key is stored, $\ell_K$ is the length of the key (such that $0 \leq K$ and $K + \ell_K \leq \ell_R$), $I$ is the position in the RAM where the input $x$ is put, $\ell_I$ is the length of the input (such that $0 \leq I$ and $I + \ell_I \leq \ell_R$), $G$ is the position in the RAM where the instructions are put, $\ell_G$ is the number of instructions (such that $0 \leq G$ and $G + \ell_G \leq \ell_R$), $O$ is the position in the RAM where the output $y$ is to be put, $\ell_O$ is the length of the output (such that $0 \leq O$ and $O + \ell_O \leq \ell_R$). Furthermore, $\mathcal{K} : \{0,1\}^* \to (\{0,1\}^w)^{\ell_K}$ parses a key into words, $\mathcal{X} : \{0,1\}^* \to (\{0,1\}^w)^{\ell_I}$ parses an input into words, $\mathcal{X}^{-1} : (\{0,1\}^w)^{\ell_I} \to \{0,1\}^*$ is a decoder such that $\mathcal{X}^{-1}\mathcal{X}x = x$, $\mathcal{Y} : (\{0,1\}^w)^{\ell_O} \to \{0,1\}^*$ takes an output represented as words and reconstructs it, $\mathcal{Y}^{-1}$ is a simulator discussed below, $g = \{g_G\}_{G \in \{0,1\}^\gamma}$ is a family of functions, for some fixed constant $\gamma$, where for each $G \in \{0,1\}^\gamma$ the function $g_G : \{0,1\}^w \times \{0,1\}^w \to \{0,1\}^w$ specifies the functionality of the instruction labelled by $G$. Each instruction is of the form $\iota_i =$

$(G_i, a_i, b_i, c_i)$, where $G_i \in \{0,1\}^\gamma$ is a *type* of an instruction and $0 \leq a_i, b_i, c_i < \ell_R$ are memory positions. All the functions should be in PPT.

We call $\mathcal{J}_K = \{K, \ldots, K + \ell_K - 1\}$ the *key positions*, $\mathcal{J}_I = \{I, \ldots, I + \ell_I - 1\}$ the *input positions*, $\mathcal{J}_G = \{G, \ldots, G + \ell_G - 1\}$ the *instruction positions*, and $\mathcal{J}_O = \{O, \ldots, O + \ell_O - 1\}$ the *output positions*. We call $\mathcal{J}_W = \{j | \exists i \in \{0, \ldots, \ell_G - 1\} \text{ s.t. } (\iota_i = (\cdot, \cdot, \cdot, j))\}$ the *intermediary positions*.

Consider the following *execution game*, taking as input a key $\mathsf{K}$ and any tuple $(x_0, \ldots, x_{\ell_I - 1}) \in (\{0,1\}^w)^{\ell_I}$, where $R[i]$ refers to the $i$-th location in the disk of the RAM.

1. Let $(\mathsf{K}_0, \ldots, \mathsf{K}_{\ell_K - 1}) = \mathcal{K}(\mathsf{K})$ and for $0 \leq i < \ell_K$ update $R[K + i] \leftarrow \mathsf{K}_i$.
2. for $0 \leq i < \ell_I$ update $R[I + i] \leftarrow x_i$.
3. In sequence for $i = 0, \ldots, \ell_P - 1$, proceed as follows: Parse $\iota_i = (G_i, a_i, b_i, c_i)$ and then update $R[c_i] \leftarrow g_{G_i}(R[a_i], R[b_i])$.
4. Let $y = \mathcal{Y}(R[O], \ldots, R[O + \ell_O - 1])$.

We make the following requirements:

**Strong Correctness:** For any $(x_0, \ldots, x_{\ell_I - 1})$ and any $\mathsf{K}$ let $y$ be computed as in the execution game. Then it is always the case that $y = \mathcal{G}_{\mathsf{K}}(\mathcal{X}^{-1}(x_0, \ldots, x_{\ell_I - 1}))$. Note that for $(x_0, \ldots, x_{\ell_I - 1}) = \mathcal{X}(x)$ we have that $\mathcal{X}^{-1}(x_0, \ldots, x_{\ell_I - 1}) = x$ such that $y = \mathcal{G}_{\mathsf{K}}(x)$. However, here we need the stronger property where we do not assume that $(x_0, \ldots, x_{\ell_I - 1}) = \mathcal{X}(x)$ for some $x$.

**Output Simulatability:** For any $(x_0, \ldots, x_{\ell_I - 1})$ and any $\mathsf{K}$ let $y = \mathcal{G}_{\mathsf{K}}(\mathcal{X}^{-1}(x_0, \ldots, x_{\ell_I - 1}))$. Then it holds that the random variable corresponding to $(R[O], \ldots, R[O + \ell_O - 1])$ in a random run of the execution game and the random variable corresponding to $\mathcal{Y}^{-1}((x_0, \ldots, x_{\ell_I - 1}), y)$ have the same distribution. The reason for this requirement is that we want to avoid that the representation of the output leaks anything extra to the output. Hence we require that the adversary could compute the representation from just the output. The reason why we give $(x_0, \ldots, x_{\ell_I - 1})$ as input to the simulator is that it does not hurt, as the adversary already knows this value and it might give a more liberal definition.

**Write before read:** In the execution game, the RAM never reads a position which was not written.

**Don't overwrite:** $|\mathcal{J}_K \cup \mathcal{J}_I \cup \mathcal{J}_G \cup \mathcal{J}_W| = \ell_K + \ell_I + \ell_G + \ell_G$. This implies that the program never overwrites positions of the key, input or instructions and never writes the same intermediary position twice.

**Reserve nought:** $0 \notin \mathcal{J}_K \cup \mathcal{J}_I \cup \mathcal{J}_G \cup \mathcal{J}_W \cup \mathcal{J}_O$. This implies that the starting location, namely the 0-th one, is *reserved* for some special purpose (to be specified later).

**Reserved tokens:** For all instructions $\iota_i$ we assume that

$$G_i \notin \{\texttt{load input}, \texttt{lift key}, \texttt{reveal output}, \texttt{done}\}.$$

This means that the labels specifying the type of an instruction cannot be of the above reserved type (which will be used to serve specific operational purposes).

**Constant Fan-Out:** There exists a constant $\alpha$ such that in the execution game, the RAM never reads a given position more than $\alpha$ times.

It is easy to verify that the above is without loss of generality, and that one can construct such a regular program for all functions.

## 5.2 The Compiled Program

The main idea behind the compiler is to store the program and all the intermediary values on the secret disk.[15] Each value $V$ on the disk will be stored along with some augmentation.

Specifically, in the preprocessing a uniformly random string $L \in \{0,1\}^\kappa$, that we call the *label* from now on, is chosen and stored in position 0 on the secret disk. All other values $V$ will have a type $(j, L, a, p, V)$, which we call *augmented value*. Here, $j$ is the position at which the value is stored, i.e., $\omega_1^{\mathsf{h}}[j] = (j, L, a, p, V)$, $L$ is the secret label, $a$ is the value of the activation counter `ac` and $p$ is the value of the program counter `pc` when $V$ was written, and $V$ is the value itself. Adding the secret label $L$ (unknown to the adversary $\mathcal{A}$) to the augmented value prevents the adversary from using the `REPLACE` command to write anything of the form $(\cdot, L, \cdot, \cdot, \cdot)$ to the secret disk. Hence all such values are from the pre-processing, or computed and stored by the CPU. Having $j$ in the augmented value prevents the adversary from using the `COPY` command to move an augmented value of the form $(j, L, \cdot, \cdot, \cdot)$ to another memory position, as we will ask the CPU to check that $j$ matches the position from which the augmented value was read.

This means that the attack possibilities of the adversary are reduced to replacing a value $(j, L, \cdot, \cdot, V)$ at memory position $j$ with an older value of the form $(j, L, \cdot, \cdot, V')$. By adding $a$ and $p$ to the augmentations we can allow the CPU to detect such a *reset attack* as follows: We ensure that for every $a$ and $p$ the CPU writes a *unique* value, of the form $(j, L, a, p, \cdot)$, only *once*. This way the CPU can use $j$ and the current values of the counters `ac` and `pc` to recover the values $a$ and $p$, and check that these values match with the values of $a$ and $p$ stored in $\omega_1^{\mathsf{h}}[j]$. The above ensures that each value $V$ occurring at $\omega_1^{\mathsf{h}}[j] = (j, L, a, p, V)$ is the correct value $V$ for memory position $j$ and the current activation and program step. This property is maintained inductively.

The compiler $\mathbf{C}_{\mathsf{h}}$ is given in Fig. 8–9. Recall that each instruction has a type $(\mathsf{Y}^{\mathsf{h}}, \mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$ (cf. Definition 3); we call $(\mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}})$ the *IO pattern* of the instruction. Algorithm $\mathsf{Compute}$ is given in Fig. 10–12. As our hybrid CPU is deterministic, $\mathsf{Random}$ always outputs the empty string. In the description self-destruct means set all registers to 0, raise the self-destruct flag, and terminate. A few remarks are in order:

- **Writing back the label.** Notice that, in each execution, the CPU not only reads the label from the location $(1, 0)$ (i.e., position number 0 of the secret disk) but also writes the label back into location $(1, 0)$ afterwards. This is necessary to ensure that the same content at $(0, 1)$ is not being read "too many" times (which is achieved by the $c$-bounding property). Whenever the CPU writes into $(1, 0)$, irrespective of the actual value, the content is overwritten. (Recall that in order to protect against replacing attacks the secret label has to match at each CPU execution.) Therefore, overwriting each time ensures that the same value is being read at most *once*.

[15] Note that the program is stored both in the secret and in the public disk. On the one hand, it is important to keep it in the public disk as otherwise the simulator could never know the current location of the secret disk which is being accessed at a certain time. On the other hand, it is necessary to store the program into the secret disk as well, so that tampering the program in the public disk will be detected.

**Load Input:** For $i = 0, \ldots, \ell_I - 1$, set the instruction at position $\mathtt{pc} = i$ as

$$\omega_0^{\mathsf{h}}[\mathtt{pc}] := (\mathtt{load\ input}, ((1,0),(0,I+i)),((1,0),(1,I+i)))\ .$$

The purpose of this instruction will be to move the $i$-th word of the input from the public disk to the secret disk. Note that the instruction reads at $\omega_1^{\mathsf{h}}[0]$ to get the label $L$ used to store the input in the correct augmented form.

**Lift Key:** For $i = 0, \ldots, \ell_K - 1$, set the instruction at position $\mathtt{pc} = \ell_I + i$ as

$$\omega_0^{\mathsf{h}}[\mathtt{pc}] := (\mathtt{lift\ key}, ((1,0),(1,K+i)),((1,0),(1,K+i)))\ .$$

The purpose of this instruction will be to increment the activation value associated the $i$th word of the secret key on the secret disk.

**Compute:** For $i = 0, \ldots, \ell_G - 1$, let $\iota_i = (G_i, a_i, b_i, c_i)$. Then set the instruction at position $\mathtt{pc} = \ell_I + \ell_K + i$ as

$$\omega_0^{\mathsf{h}}[\mathtt{pc}] := (G_i, ((1,0),(1,a_i),(1,b_i),(1,j)),((1,0),(1,c_i),(1,j)))\ ,$$

where $j = G + i$. The purpose of this instruction will be to execute instruction number $i$. Note that the instruction reads at $\omega_1^{\mathsf{h}}[G+i]$, as here we store a copy of the instruction $(G_i, a_i, b_i, c_i)$ (to detect tampering of the public disk).

**Reveal Output:** For $i = 0, \ldots, \ell_O - 1$, set the instruction at position $\mathtt{pc} = \ell_I + \ell_K + \ell_G + i$ as

$$\omega_0^{\mathsf{h}}[\mathtt{pc}] := (\mathtt{reveal\ output}, ((1,0),(1,i)),((1,0),(0,i)))\ .$$

The purpose of this instruction will be to move the $i$th word of the output to the public disk.

**Done:** Let $\mathtt{pc} = \ell_I + \ell_K + \ell_G + \ell_O$. Set

$$\omega_0^{\mathsf{h}}[\mathtt{pc}] := (\mathtt{done}, ((1,0),(1,0)))\ .$$

This is a sentinel instruction.

**Figure 8:** The Compiler, Public Disk.

- **Updating counters.** Notice that we update the activation counter (`ac`) *only* in the following cases:

  1. `load input` (c.f. Fig. 10): When $a = \mathtt{ac} - 1$ and $\mathtt{pc} = 0$. This is the first time the CPU loads the label in one activation, and therefore we have to update the value corresponding to the activation counter inside the augmented encoding of the label. Once it is updated there is no more change, as for $\mathtt{pc} > 0$ the CPU ensures that $a = \mathtt{ac}$.
  2. `lift key` (c.f. Fig. 10): This is done because, before using the key, we have to update the value of the activation counter in the augmented encoding of the key.
  3. `compute` G (c.f. Fig. 11): Whenever some instruction with type G is computed, the activation counter is increased in the augmented encoding of the instruction stored in the secret disk; this ensures that each instruction is executed only once in one activation.

  The program counter `pc` is updated only inside the augmented encoding of the label, during

**Label the Parameters:** Pick a uniformly random *label* $L \in \{0,1\}^\kappa$ for $\kappa = k+1$. Set

$$\omega_1^{\mathsf{h}}[0] := (0, L, -1, -1, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \ .$$

**Key:** Let $(\mathsf{K}_0, \ldots, \mathsf{K}_{\ell_K - 1}) = \mathcal{K}(\mathsf{K})$ and for $0 \le i < \ell_K$, set the value at position $j = K + i$ as

$$\omega_1^{\mathsf{h}}[j] := (j, L, -1, \ell_I + i, \mathsf{K}_i) \ .$$

**Compute:** For $i = 0, \ldots, \ell_G - 1$, let $\iota_i = (G_i, a_i, b_i, c_i)$, set the value at position $j = G + i$ as

$$\omega_1^{\mathsf{h}}[j] := (j, L, -1, \ell_I + \ell_K + i, (G_i, a_i, b_i, c_i)) \ .$$

This value will be used to verify the instruction from the public disk (to detect tampering).

**Figure 9:** The Compiler, Secret Disk.

each execution of the CPU; this follows by the fact that the CPU accesses the label whenever run, and accesses all the other values once.

- **Implicit checking of the label.** Note that, in the description of $\mathsf{Compute}$, there is no explicit check of the labels. The check is done implicitly by attempting to parse the content of the input registers in the specified format. For example, in case the CPU parses two registers successfully as $(\cdot, L, \cdot, \cdot, \cdot)$ and $(\cdot, L, \cdot, \cdot, \cdot)$, then the label in the two registers must match.

Next we state the following theorem about $\mathsf{HS}$.

**Theorem 5.** *The hybrid-scheme* $\mathsf{HS}$ *is an* $O(1)$*-bounding, extra-space insensitive hybrid scheme.*

We turn to a high-level overview of the security proof (cf. Section 5.3 for a formal proof). Our goal is to prove that the above hybrid scheme is RAM-simulatable, namely for all adversaries $\mathcal{B}$ attacking the hybrid scheme in a hybrid execution, there exists a simulator $\mathcal{S}$ faking the view of $\mathcal{B}$ only given black-box access to the original functionality $\mathcal{G}_{\mathsf{K}}$.

As a first step, we prove that the probability by which the adversary succeeds in using a "replace" command to write some value on the secret disk with the correct secret label, and having the CPU read this value without provoking a self-destruct, is essentially equal to the probability of guessing the secret label (which is exponentially small). This means we can assume that all the values put on the secret disk using a "replace" command does not contain the secret label. In each execution our CPU $\mathsf{CPU}^{\mathsf{h}}$ will check that all loaded values contain the same label, and will write back values where the augmentation contains this label. It then follows that all values with the secret label in the augmentation were written by the pre-processing or $\mathsf{CPU}^{\mathsf{h}}$, and it also follows that all values not having the secret label in the augmentation are known by the adversary: They were put on disk using a `REPLACE` command or computed by $\mathsf{CPU}^{\mathsf{h}}$ on values known by the adversary. We then argue that $\mathsf{CPU}^{\mathsf{h}}$ (by design) will never write two values $V \neq V'$ sharing the same augmentation $(j, L, a, p)$. This is because it includes the strictly increasing $(a, p)$ in the augmentation, and we also prove that $\mathsf{CPU}^{\mathsf{h}}$ can predict what $(a, p)$ should be in all loaded values in all executions. It then follows from an inductive argument that all values containing the secret label in the augmentation are correct. Hence all values on the secret disk are either correct secret values or incorrect values known by the adversary. So, when $\mathsf{CPU}^{\mathsf{h}}$ writes a result to the public disk, it is either an allowed

$\mathsf{Y}^{\mathsf{h}} = \texttt{load input:}$ If not $(\mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}}) = ((1,0),(0,j)),((1,0),(1,j))$, for some $j$, then self-destruct. Try to parse the input values as follows

$$(0, L, a, \mathtt{pc} - 1, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \leftarrow \mathtt{R}_1^{\mathsf{h}} \quad \textit{// read label from } \mathcal{SD}_1^{\mathsf{h}}$$
$$x \leftarrow \mathtt{R}_2^{\mathsf{h}} \quad \textit{// read input from } \mathcal{PD}^{\mathsf{h}}$$

If the parsing or any of the following tests fail, then self-destruct:

- $0 \le \mathtt{pc} < \ell_I$ *// CPU supposed to load input*
- $j = I + \mathtt{pc}$ *// correct position is read*
- $a = \mathtt{ac}$ or $(a = \mathtt{ac} - 1$ and $\mathtt{pc} = 0)$ *// counters are consistent*

Otherwise, set

$$\mathtt{O}_1^{\mathsf{h}} \leftarrow (0, L, \mathtt{ac}, \mathtt{pc}, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \quad \textit{// write label back to } \mathcal{SD}_1^{\mathsf{h}}$$
$$\mathtt{O}_2^{\mathsf{h}} \leftarrow (j, L, \mathtt{ac}, \mathtt{pc}, x) \quad \textit{// store input in augmented form}$$

$\mathsf{Y}^{\mathsf{h}} = \texttt{lift key:}$ If not $(\mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}}) = ((1,0),(1,j)),((1,0),(1,j))$, for some $j$, then self-destruct. Try to parse the input values as follows

$$(0, L, \mathtt{ac}, \mathtt{pc} - 1, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \leftarrow \mathtt{R}_1^{\mathsf{h}} \quad \textit{// read label from } \mathcal{SD}_1^{\mathsf{h}}$$
$$(j, L, \mathtt{ac} - 1, \mathtt{pc}, z) \leftarrow \mathtt{R}_2^{\mathsf{h}} \quad \textit{// read key from } \mathcal{SD}_1^{\mathsf{h}}$$

If the parsing or any of the following tests fail, then self-destruct:

- $\ell_I \le \mathtt{pc} < \ell_I + \ell_K$ *// CPU supposed to lift key*
- $j = K + p$, where $p = \mathtt{pc} - \ell_I$ *// correct position is read*

Otherwise, set

$$\mathtt{O}_1^{\mathsf{h}} \leftarrow (0, L, \mathtt{ac}, \mathtt{pc}, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \quad \textit{// write back label to } \mathcal{SD}_1^{\mathsf{h}}$$
$$\mathtt{O}_2^{\mathsf{h}} \leftarrow (j, L, \mathtt{ac}, \mathtt{pc}, z) \quad \textit{// write key to } \mathcal{SD}_1^{\mathsf{h}} \textit{ (after updating } \mathtt{ac}\textit{)}$$

**Figure 10:** Compute, Part I.

output or a value already known by the adversary. From the above intuition, it is straight-forward although rather tedious to derive a simulator.

### 5.3 Analysis

We imagine the values in the secret disks being partitioned into so-called *domains*, labeled by $D \in \{0,1\}^{\kappa}$. If a value of the form $(j, D, \ldots)$ is stored into the secret disk, then we say that it is stored in the domain $D$. Notice that the pre-processing stores all the values in the domain $L$ for a uniformly random $L$ that is unknown to the adversary. We call this particular domain indexed by the secret label $L$ the *secret domain*.

In the following analysis, we are going to use the following crucial *properties* of $\mathsf{CPU}^{\mathsf{h}}$. One can easily verifies that these properties hold by the construction of Compute (c.f. Fig. 10–12):

$\mathsf{Y}^{\mathsf{h}} = \mathrm{G} \notin \{\texttt{load input}, \texttt{lift key}, \texttt{reveal output}, \texttt{done}\}$:

If not $(\mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}}) = ((1,0),(1,a_i),(1,b_i),(1,j)),((1,0),(1,c_i),(1,j))$, for some $a_i, b_i, j, c_i$, then self-destruct. Try to parse the input values as follows

$$(0, L, \mathtt{ac}, \mathtt{pc}-1, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \leftarrow \mathtt{R}_1^{\mathsf{h}} \quad \textit{// read label from } \mathcal{SD}_1^{\mathsf{h}}$$
$$(a_i, L, \mathtt{ac}, \cdot, A) \leftarrow \mathtt{R}_2^{\mathsf{h}} \quad \textit{// read input from } \mathcal{SD}_1^{\mathsf{h}}$$
$$(b_i, L, \mathtt{ac}, \cdot, B) \leftarrow \mathtt{R}_3^{\mathsf{h}} \quad \textit{// read input data from } \mathcal{SD}_1^{\mathsf{h}}$$
$$(j, L, \mathtt{ac}-1, \mathtt{pc}, H) \leftarrow \mathtt{R}_4^{\mathsf{h}} \quad \textit{// read instruction from } \mathcal{SD}_1^{\mathsf{h}}$$

If the parsing or any of the following tests fail, then self-destruct:

- $\ell_I + \ell_K \leq \mathtt{pc} < \ell_I + \ell_K + \ell_G$ *// CPU supposed to compute*
- $j = G + p$, where $p = \mathtt{pc} - \ell_I - \ell_K$ *// instruction read from correct position*
- $H = (\mathrm{G}, a_i, b_i, c_i)$ *// instruction at $\mathcal{PD}$ matches the one read from $\mathcal{SD}_1^{\mathsf{h}}$*

Otherwise, set $C := g_{\mathrm{G}}(A, B)$; and let *// compute according to the* G*-type*

$$\mathtt{O}_1^{\mathsf{h}} \leftarrow (0, L, \mathtt{ac}, \mathtt{pc}, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) = \quad \textit{// write back label to } \mathcal{SD}_1^{\mathsf{h}}$$
$$\mathtt{O}_2^{\mathsf{h}} \leftarrow (c_i, L, \mathtt{ac}, \mathtt{pc}, C) \quad \textit{// write computed value to } \mathcal{SD}_1^{\mathsf{h}}$$
$$\mathtt{O}_3^{\mathsf{h}} \leftarrow (j, L, \mathtt{ac}, \mathtt{pc}, H) \quad \textit{// write back instruction to } \mathcal{SD}_1^{\mathsf{h}} \textit{ (updating counters)}$$

**Figure 11:** Compute, Part II.

1. If in a single execution the values read from the secret disk $\mathcal{SD}_1^{\mathsf{h}}$ by the CPU do not come from the same domain, then the CPU self-destructs.

2. If in a single execution all the values read from the secret disk by the CPU are from the same domain $D$, then all the values written to the secret disk by the CPU in that activation are written to domain $D$.

3. The values that the CPU writes to the public disk $\mathcal{PD}^{\mathsf{h}}$ do not depend on the domain it reads from $\mathcal{SD}_1^{\mathsf{h}}$ More precisely, changing *only* the domain $D$ in all the values read from the secret disk to some $D' \neq D$ will *not* change the value CPU writes into $\mathcal{PD}^{\mathsf{h}}$.

We do a hybrid proof, starting from the hybrid scheme and ending with the ideal world. We prove a series of technical lemmas, and in between we use the insight of the lemma to define an indistinguishable hybrid distribution. We will define the simulator, which simulates the view of the modified adversary only given the access to the ideal functionality, at the end of this sequence of hybrid distributions.

Let $\mathcal{B}$ be any adversary attacking our hybrid scheme in the game $\textsc{Hyb}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$. We are going to define the event MIX that happens when the adversary $\mathcal{B}$ uses the `REPLACE` command to write a value into the secret domain *and* this value is subsequently read by $\mathsf{CPU}^{\mathsf{h}}$. More precisely, it gives a `REPLACE` command with a value of the form $(\cdot, L, \cdots)$ for the value $L$ placed as the secret label in the pre-processing and later the CPU reads from a position where that particular $(\cdot, L, \cdots)$ is stored. We define DESTRUCT to be the event that the CPU self-destructs.

$\mathsf{Y}^{\mathsf{h}} = \texttt{reveal output:}$ If not $(\mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}}) = ((1,0),(1,j)),((1,0),(0,j))$, for some $j$, then self-destruct. Try to parse the input values as follows

$$(0, L, \mathtt{ac}, \mathtt{pc} - 1, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \leftarrow \mathtt{R}_1^{\mathsf{h}} \quad \textit{// read label from } \mathcal{SD}_1^{\mathsf{h}}$$
$$(j, L, \mathtt{ac}, \cdot, y) \leftarrow \mathtt{R}_2^{\mathsf{h}} \quad \textit{// read output from } \mathcal{SD}_1^{\mathsf{h}}$$

If the parsing or any of the following tests fail, then self destruct:

- $\ell_I + \ell_K + \ell_G \leq \mathtt{pc} < \ell_I + \ell_K + \ell_G + \ell_O$; *// CPU supposed to reveal output*
- $j = O + p$, where $p = \mathtt{pc} - \ell_I - \ell_K - \ell_G$ *// position is consistent*

Otherwise, set

$$\mathtt{O}_1^{\mathsf{h}} \leftarrow (0, L, \mathtt{ac}, \mathtt{pc}, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \quad \textit{// write back label to } \mathcal{SD}_1^{\mathsf{h}}$$
$$\mathtt{O}_2^{\mathsf{h}} \leftarrow y \quad \textit{// write output to } \mathcal{PD}^{\mathsf{h}} \textit{ (remove augmentation).}$$

$\mathsf{Y}^{\mathsf{h}} = \texttt{done:}$ If not $(\mathsf{I}^{\mathsf{h}}, \mathsf{O}^{\mathsf{h}}) = ((1,0)),((1,0))$, for some $j$, then self-destruct. Try to parse the input values as follows

$$(0, L, \mathtt{ac}, \mathtt{pc} - 1, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \leftarrow \mathtt{R}_1^{\mathsf{h}} \quad \textit{// read label from } \mathcal{SD}_1^{\mathsf{h}}$$

If the parsing or any of the following tests fail, then self destruct:

- $\mathtt{pc} = \ell_I + \ell_K + \ell_G + \ell_O$. *// CPU supposed to terminate*

Otherwise, set

$$\mathtt{O}_1^{\mathsf{h}} \leftarrow (0, L, \mathtt{ac}, \mathtt{pc}, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \quad \textit{// write back label to } \mathcal{SD}_1^{\mathsf{h}}$$

and end the activation.

**Figure 12:** Compute, Part III

**Lemma 6.** *The probability of* MIX $\wedge \neg$DESTRUCT *is at most* $2^{-\kappa+1}$ *irrespective of the strategy of the adversary* $\mathcal{B}$.

*Proof.* We first argue that it is sufficient to prove that as long as MIX and DESTRUCT did not occur the probability that MIX occurs in the next command and DESTRUCT does not is at most $2^{-\kappa+1}$. Then we prove this fact.

Assume we can prove that as long as MIX and DESTRUCT did not occur the probability that MIX $\wedge \neg$DESTRUCT occurs in the next step is at most $2^{-\kappa+1}$. Consider then the *first* step in which MIX occurs. Then clearly MIX did not occur in an earlier step. Also, since the step was reached, we have that DESTRUCT did not occur in an earlier step either. So, in all earlier steps, MIX and DESTRUCT did not occur. So, we know that in the current step, the probability that MIX $\wedge \neg$DESTRUCT occurs is at most $2^{-\kappa+1}$. However, by assumption, we know that MIX occurs, so either MIX $\wedge \neg$DESTRUCT or MIX $\wedge$ DESTRUCT will occur. If MIX $\wedge$ DESTRUCT occurs, then there will be no more steps, as we self destruct. Hence, the event MIX $\wedge \neg$DESTRUCT occurs the first time where MIX occurs, or never. And, the first time MIX occurs, the event MIX$\wedge\neg$DESTRUCT occurs with probability at most $2^{-\kappa+1}$, as argued above.

By MIX and DESTRUCT not occurring and Property 1 above, we see that the CPU only ever

read values from the same domain in a single execution. So, by Property 2, the CPU never in a single invocation read values from the secret domain and then stored to another domain. Since the CPU has no memory between invocations it follows that no information about $L$ was ever written to a domain which is not the secret domain. So, by additionally using Property 3, we see that as long as $\mathsf{MIX}$ and $\mathsf{DESTRUCT}$ did not occur, the adversary has no information on $L$. Since the guessing probability of $L$ starts out at $2^{-\kappa}$ right after the pre-processing, it follows that as long as $\mathsf{MIX}$ and $\mathsf{DESTRUCT}$ did not occur, the guessing probability of $L$ in the view of the adversary is at least $2^{-\kappa}$.

There are, however, other ways that adversary can learn information about $L$. It can write a value $(j', L', \ldots)$ to some position $j'$ on the secret disk and then execute the CPU on position $j'$ and some position $j$ where a value $(j, L, \ldots)$ is stored which contains the secret label $L$. We call this a *mixing attack*. By property 1 above, if the CPU does not self-destruct after a mixing attack, then $L' = L$, so the one bit of information about whether the CPU self-destructs or not will depend on $L$.

It is not hard to see that if the adversary performs a mixing attack, then the event $\mathsf{MIX}$ occurs. We now analyse the development of the average guessing probability of $L$ after mixing attacks. We first look at what happens at the first mixing attack and we do a case analysis on $L' = L$ and $L' \neq L$. Let $g$ denote the guessing probability of $L$ in the view of the adversary before the first mixing attack.

If $L' = L$ in a mixing attack, then the adversary guessed $L$ as $L' = L$ occurs in the command $(j', L', \ldots)$, so this case occurs with probability at most $g$. After this event the average guessing probability is clearly 1. This can clearly never become higher.

If $L' \neq L$ in a mixing attack, then the adversary ruled out the value $L'$. For all labels $L''$, let $p(L'')$ be the probability in the view of the adversary that $L = L''$ before the mixing attack and let $q(L'')$ be the probability in the view of the adversary that $L = L''$ after the mixing attack. We have that $q(L') = 0$ and $q(L'' \neq L') = p(L'')/(1 - p(L'))$. Hence $\max_{L''} q(L'') \leq \max_{L''} p(L'')/(1 - p(L')) = g(1 - p(L'))^{-1} \leq g(1 - g)^{-1}$.

It follows that after the first mixing attack it holds that either (with probability $g$) the adversary can guess $L$ with probability 1 or (with probability at most $(1 - g)$) the adversary can guess $L$ with probability at most $g(1 - g)^{-1}$. So, by the law of total probability, after the first mixing attack the probability that the adversary can guess $L$ is at most $g \cdot 1 + (1 - g) \cdot g(1 - g)^{-1} = 2g$.

Note then that this probability cannot increase further by making further mixing attacks. Namely, if the first mixing attack is with $L' \neq L$, then the CPU self-destructs, and hence no further mixing attacks are possible. And, if $L' = L$, then the adversary has just learned $L$ and the guessing probability increased to its maximal value 1.

This means that the average guessing probability of $L$ in the view of the adversary after mixing attacks is $2g = 2^{1-\kappa}$. □

Now, for an adversary $\mathcal{B}$ attacking $\mathsf{HS}$, let $\mathcal{B}_1$ be a corresponding adversary which runs exactly as $\mathcal{B}$ except that it never uses the `REPLACE` command to the secret disk. Instead, $\mathcal{B}_1$ internally keeps track of the effect the `REPLACE` commands would have had on $\mathcal{SD}_1^{\mathsf{h}}$. In doing this, it keeps track of the record $\mathsf{S}$ [16] kept by the game $\textsc{Hyb}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$ – it is easy to see that $\mathcal{B}$ could efficiently compute an exact copy of $\mathsf{S}$. Now from the definition of $\mathsf{S}$, we observe that if $\mathsf{S}[j] = \perp$, then the value in $D_1^{\mathsf{h}}[j]$ is efficiently computable by $\mathcal{B}$ as either (1) it was put in $D_1^{\mathsf{h}}[j]$ by a `REPLACE` command, (2)

[16] Recall that $\mathsf{S}[i]$ keeps track of the fact if the current value at location $i$ (in the secret disk) is still 'unknown' to the adversary, and if it is then the sequence number indicating the order when it is stored.

1. Initially, let $\mathsf{S}[i] = \top$ for all $i$, except that $\mathsf{S}[0] = \bot$, and $\mathsf{S}[j] = \bot$ for all key positions $j$ and all code positions $j$. We use $\mathsf{S}[i] = \top$ to represent $\mathsf{S}[i] \neq \bot$, as we are not interested in the exact value of $\mathsf{S}[i]$ when $\mathsf{S}[i] \neq \bot$.
2. Receive $P$ and $\Omega$ and input them to $\mathcal{B}$. Then run $\mathcal{B}$.
3. When $\mathcal{B}$ outputs $(\texttt{STOP}, z)$, then output $(\texttt{STOP}, z)$.
4. When $\mathcal{B}$ outputs $(\texttt{REPLACE}, (1, j, z))$, then update $V[j] \leftarrow z$ and $\mathsf{S}[j] \leftarrow \bot$.
5. When $\mathcal{B}$ outputs $(\texttt{REPLACE}, (0, j, z))$, then output $(\texttt{REPLACE}, (0, j, z))$
6. When $\mathcal{B}$ outputs $(\texttt{COPY}, j, j')$, update $\mathsf{S}[j'] \leftarrow \mathsf{S}[j]$ and if $\mathsf{S}[j] = \bot$, then $V[j'] \leftarrow V[j]$.
7. When $\mathcal{B}$ makes the command $(\texttt{EXEC})$, then let $D_0$ be the current public disk and inspect $D_0[\mathsf{pc}]$ to get the instruction which is about to be executed. Suppose, for this execution, $(i_0, \ldots, i_g)$ be the positions to be read on the secret disk and $(j_0, \ldots, j_h)$ be the positions to be written on the secret disk.
   (a) If $\mathsf{S}[i_0] = \cdots = \mathsf{S}[i_g] = \bot$, which means all the values to be read are 'known' (i.e. this is a public execution), then proceed as follows:
      - Use $V[i_0], \ldots, V[i_g]$ and the values read from the public disk by the current instruction $D_0[\mathsf{pc}]$, $\mathtt{ac}$ and $\mathtt{pc}$ to compute the values to store into the output registers $\mathtt{O}^{\mathsf{h}}_1, \ldots, \mathtt{O}^{\mathsf{h}}_d$ which would be obviously exactly the same values that $\mathsf{CPU}^{\mathsf{h}}$ would have computed on $(\texttt{EXEC})$.
      - Update $\mathsf{S}$ by setting $\mathsf{S}'[j_0] = \cdots = \mathsf{S}'[j_h] := \bot$.
      - Similarly, update $V$ as follows: for $a = 0, \ldots, h$, we update $V[j_a]$ to hold the value that $\mathsf{CPU}^{\mathsf{h}}$ would have written to $D_1[j_a]$ namely $V[j_a] := \mathtt{O}^{\mathsf{h}}{}_a$
      - Finally, for all positions in $\mathcal{PD}^{\mathsf{h}}$, (if any) where $\mathsf{CPU}^{\mathsf{h}}$ would have written, compute the value and use $\texttt{REPLACE}$ command to write (since replacing on $\mathcal{PD}^{\mathsf{h}}$ is allowed).
   (b) Otherwise, this is a secret execution and we proceed as follows:
      i. If $\mathsf{S}[i_0] \neq \bot, \ldots, \mathsf{S}[i_g] \neq \bot$, then output $(\texttt{EXEC})$. Then set $\mathsf{S}[j_1] = \cdots = \mathsf{S}[j_h] := \top$.
      ii. Otherwise, there exists $a$ and $b$ such that $\mathsf{S}[i_a] \neq \bot$ and $\mathsf{S}[i_b] = \bot$. In that case, simulate a self destruct (by ignoring this and all future $\texttt{EXEC}$ commands).

**Figure 13:** $\mathcal{B}_1$

copied to $D^{\mathsf{h}}_1[j]$ from $D^{\mathsf{h}}_1[i]$ where $D^{\mathsf{h}}_1[i]$ was computable, or (3) computed by the CPU by a run of $\mathsf{Compute}$ on inputs which were all computable (notice that our CPU is deterministic). So, $\mathcal{B}$ can efficiently compute $D^{\mathsf{h}}_1[j]$ for all $j$ where $\mathsf{S}[j] = \bot$. Now, for all $j$ where $\mathsf{S}[j] = \bot$ we will let $\mathcal{B}_1$ compute the value which would have been in $D^{\mathsf{h}}_1[j]$, had the $\texttt{REPLACE}$ commands of $\mathcal{B}$ been executed. Additionally, $\mathcal{B}_1$ will make sure that for all $j$ where $\mathsf{S}[j] \neq \bot$, it keeps in $D^{\mathsf{h}}_1[j]$ the value that would have been in $D^{\mathsf{h}}_1[j]$ if the $\texttt{REPLACE}$ commands of $\mathcal{B}$ had been executed. $\mathcal{B}_1$ maintains another record called $V$ to store the known values – that is whenever it sets $\mathsf{S}[j]$ to $\bot$, it updates $V[j]$. The details are given in Figure 13.

Let $E$ be an event defined in $\mathcal{B}$ and $\mathcal{B}_1$, there exist two positions $j, j'$ in $D^{\mathsf{h}}_1$ from which $\mathsf{CPU}^{\mathsf{h}}$ reads in an execution such that the following happens: $\mathsf{S}[j] = \bot$ and $\mathsf{S}[j'] \neq \bot$. Notice that, for $\mathcal{B}_1$, $E$ happens in Step 7(b)ii. By construction, if $E$ does not occur, then $\mathcal{B}_1$ and $\mathcal{B}$ will output exactly the same value. However, until $E$ occurs it holds for all values $D^{\mathsf{h}}_1[j]$ where $\mathsf{S}[j] \neq \bot$ that $D^{\mathsf{h}}_1[j]$ is a

value put by the pre-processing (alternatively a copied or computed from such values). That is, if $\mathsf{S}[j] \neq \perp$ then $D_1^{\mathsf{h}}[j]$ is in the secret domain $L$. On the other hand $\mathsf{S}[j'] = \perp$ implies that the content of $D_1^{\mathsf{h}}[j']$ is known to the adversary and hence the value is in the public domain $L'$ . Hence, when $E$ occurs, we have that $\mathsf{CPU}^{\mathsf{h}}$ reads from two domains $L$ and $L'$. So, in that case, by Property-1 of $\mathsf{CPU}^{\mathsf{h}}$, DESTRUCT occurs if $L \neq L'$. On the other hand, if $L = L'$ the CPU does not self-destruct in $\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)$, then MIX occurs, which in turn implies that $\mathsf{MIX} \wedge \neg\mathsf{DESTRUCT}$ occurred. Using Lemma 6 we can conclude that:

$$\left\{\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} \approx_{2^{-k+1}} \left\{\text{HYB}_{\mathsf{HS},\mathcal{B}_1,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} .$$

In the following we can therefore assume an adversary $\mathcal{B}_1$ which does *not* use the `REPLACE` command on the secret disk.

**Lemma 7.** *Let $L$ be the secret label. Assume an adversary of the form $\mathcal{B}_1$. Then for all $(j, L, a, p)$ there can* not *exist two different values $V$ and $V'$ $(\neq V)$ such that both $(j, L, a, p, V)$ and $(j, L, a, p, V')$ are written to the secret disk $\mathcal{SD}_1^{\mathsf{h}}$.*

*Proof.* By assumption, the adversary $\mathcal{B}_1$ never replaces values on the secret disk, it at most copies them around. By construction, all the values of the form $(j, L, a, p, \cdot)$ written by the CPU has $a = \texttt{ac}$ and $p = \texttt{pc}$ and these change between invocations of the CPU. So, the only way two values of the form $(j, L, a, p, V)$ and $(j, L, a, p, V' \neq V)$ could be written is that it happens in the same invocation of the CPU. In the following we use the requirements of the regular program (c.f. Section 5.1 [17]. Now considering a single execution of $\mathsf{CPU}^{\mathsf{h}}$ case by case:

1. `load input`: $\mathsf{CPU}^{\mathsf{h}}$ writes in locations 0 and $j$ but we have $j \neq 0$ as $j \geq I > 0$ by *Reserve Nought*
2. `lift key`: $\mathsf{CPU}^{\mathsf{h}}$ writes in locations 0 and $j$ but we have $j \neq 0$ as $j > 0$ by *Reserve Nought* again.
3. `compute` G: $\mathsf{CPU}^{\mathsf{h}}$ writes to locations $0, c_i, j$. Here we have $c_i \neq 0$ and $j \neq 0$ by *Reserve Nought*. Furthermore, $c_i \neq j$ by *No Overlap*.
4. `done`: In `Done` we only write one value to the secret disk.

□

Now, for an adversary $\mathcal{B}_1$ attacking HS without `REPLACE` commands to the secret disk, let $\mathcal{B}_2$ be a corresponding adversary which runs exactly as $\mathcal{B}_1$ except that it (i) ignores all `COPY` commands and (ii) ignores all `REPLACE` commands $(0, j, \cdot)$, where $j \notin \{I, \ldots, I + \ell_I - 1\}$ that is it is not allowed to replace any *pre-processed* value in $\mathcal{PD}^{\mathsf{h}}$. Instead it keeps track (1) which values currently would be where on the secret disk had the `COPY` commands been performed and (2) of the state of the public disk had the `REPLACE` commands been executed. For that, $\mathcal{B}_2$ maintains a record, namely $\widetilde{\omega_0^{\mathsf{h}}}$ corresponding to the $\omega_0^{\mathsf{h}}$ [18] (c.f. Fig. 4). When $\mathcal{B}_1$ makes an `EXEC` command, then $\mathcal{B}_2$ checks that (i)

[17] Notice that since the hybrid program depends on the regular program these requirements impose similar restriction also in the compiled program.

[18] Recall that $\omega_0^{\mathsf{h}}$ is the part, to be stored in $\mathcal{PD}^{\mathsf{h}}$, of the initial encoding output by $\mathbf{C}_{\mathsf{h}}$ in the pre-processing; precisely obtained by $(P, \omega_0^{\mathsf{h}}, \omega_1^{\mathsf{h}}) \leftarrow \mathbf{C}_{\mathsf{h}}(\mathcal{G}, \mathsf{K})$

if there is *any* replacement of the pre-processed values in $\mathcal{PD}^{\mathsf{h}}$ i.e. if $\widetilde{\omega_0^{\mathsf{h}}} \neq \omega_0^{\mathsf{h}}$ or (ii) if the current location (say $j$), to be read from $\mathcal{SD}_1^{\mathsf{h}}$, is copied, i.e. the position or the counters in the augmented value fails to match, had the `COPY` command been executed. If any of the check fails, $\mathcal{B}_2$ simulates self-destruct by ignoring all future `EXEC` commands. Otherwise it also output `EXEC`.

**Lemma 8.**

$$\left\{\textsc{Hyb}_{\mathsf{HS},\mathcal{B}_1,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} = \left\{\textsc{Hyb}_{\mathsf{HS},\mathcal{B}_2,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}}.$$

*Proof.* By the above lemma (c.f. Lemma 7), we know that for each $(j, L, a, p)$ at most one value of the form $(j, L, a, p, V)$ was ever written to the secret disk. So, if we can show that the CPU correctly predicts the correct value of $\omega_0^{\mathsf{h}}[\mathtt{pc}]$ and the correct value of $(j, L, a, p)$ for all the values $(j, L, a, p, V)$ that it reads up from the secret disk, and that it actually performs the correct checks for these values, then we know that the first time that $\textsc{Hyb}_{\mathsf{HS},\mathcal{B}_1,\mathcal{G}}(k)$ would have resulted in reading up an incorrect value $\omega_0^{\mathsf{h}}[\mathtt{pc}]$ or an incorrect $(j, L, a, p, V)$, the CPU would self-destruct, which is exactly how we simulate.

Clearly, for the labeled parameters the CPU can correctly predict that it should get something of the form $(0, L, \mathtt{ac}, \mathtt{pc} - 1)$ and indeed it self-destructs if it does not.

By construction, for $\mathtt{pc} \notin \{G, \ldots, G+\ell_G-1\}$, the values $(j, L, a, p, \ldots)$ that the CPU expects to read from the secret disk and also the instruction read from the public disk are uniquely given by `ac`, `pc`, $L$ and the (correct) parameters $(K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)$ retrieved form $\omega_1^{\mathsf{h}}[0]$. As an example, for $\mathtt{pc} \in \{K, \ldots, K+\ell_K-1\}$, it can use $K$ and $\ell_K$ to see that the label should be `lift key` and then knows that the instruction should be $(\mathtt{lift\ key}, ((1,0),(1,j)), ((1,0),(1,j)))$ for $j = K + \mathtt{pc} - \ell_I$. From this it knows that it should read a value of the form $(j, L, \mathtt{ac}-1, \mathtt{pc}, \ldots)$. Since there is at most one such value, if the CPU does not self-destruct it is because it read up *the* correct value. For $\mathtt{pc} \in \{G, \ldots, G+\ell_G-1\}$ it follows in the same manner that $j$ and $L$ follows correctly from `ac`, `pc` and $(K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)$, so the position $j$ and the expected pattern $(j, L, \mathtt{ac}-1, \mathtt{pc}, \ldots)$ follows correctly from `ac`, `pc`, $L$ and $(K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)$. Hence if the CPU does not self-destruct then $(j, L, \mathtt{ac}-1, \mathtt{pc}, H)$ is *the* correct value and hence $H = (G_i, a_i, b_i, c_i)$. Hence the check $H = (\mathrm{G}, a_i, b_i, c_i)$ guarantees that the instruction on the public disk was correct. This in turn implies that if the CPU does not self destruct then $(a_i, L, \mathtt{ac}, \cdot, A)$ and $(b_i, L, \mathtt{ac}, \cdot, B)$ are *the* correct values. Note that we here use additionally that no two values of the form $(j, L, \mathtt{ac}, \cdot, \cdot)$ are stored for $j \in \{G, \ldots, G+\ell_G-1\}$. This follows from *Don't overwrite*, which guarantees that within a single activation `ac`, no two instructions write to the same position. □

We are now looking at an adversary $\mathcal{B}_2$ which never touches the secret disk and which only replaces values in the input positions namely $\{I, \ldots, I+\ell_I-1\}$ on the public disk. Since it is predictable by $\mathcal{B}_2$ when the values in the positions $\{I, \ldots, I+\ell_I-1\}$ are used, we can replace it by a non-adaptive adversary $\mathcal{B}_3$ which is non-adaptive in a sense that it runs all the `EXEC` commands, then it runs `REPLACE` command on the input positions of the public disk $\mathcal{PD}^{\mathsf{h}}$ and then again the `EXEC` command *in sequence* rather than running them in any order. Moreover, it runs each command a specific number of times in contrast to $\mathcal{B}_2$ which runs any command for any number of times. The formal description of $\mathcal{B}_3$ is given below: as follows within each activation:

1. Execute command (`EXEC`) exactly $\ell_K$ times.
2. Execute commands $(\mathtt{REPLACE}, 0, I+i, x_i)$ for $i = 0, \ldots, \ell_I - 1$, in that sequence.

3. Execute command (EXEC) exactly $\ell_G + \ell_O + 1$ times.

The transformation from $\mathcal{B}_2$ to $\mathcal{B}_3$ works as follows:

- Initialize $p = 0$. This is a counter keeping track of how many times we "forgot" to increment pc.
- When $\mathcal{B}_2$ issues a $(\texttt{REPLACE}, 0, i, V)$ command, queue the command and don't issue any commands.
- When $\texttt{pc} + p \notin \{\ell_K, \ldots, \ell_K + \ell_I - 1\}$ and $\mathcal{B}_2$ issues an EXEC command, do the same.
- When $\texttt{pc} + p \in \{\ell_K, \ldots, \ell_K + \ell_I - 1\}$ and $\mathcal{B}_2$ issues an EXEC command, then compute $i = \texttt{pc} + p - \ell_K$ and compute the value $x_i$ that would be at position $I + i$ on the public disk if all the REPLACE commands in the queue were executed in order from the head and back. Don't issue an EXEC command, instead let $p \leftarrow p + 1$.
- When it happens after the above rule that $\texttt{pc} + p = \ell_I$, then let $(x_1, \ldots, x_{\ell_I - 1})$ be the values computed in the above rule. Then execute the commands $(\texttt{REPLACE}, 0, I + i, x_i)$ for $i = 0, \ldots, \ell_I - 1$ and then execute EXEC exactly $\ell_I$ times. Then let $p = 0$.

It is not hard to see that, by adaptivity the adversary does not gain more power which is not simulatable by a non-adaptive adversary.[19] Hence, we have that

$$\left\{ \text{HYB}_{\mathsf{HS},\mathcal{B}_2,\mathcal{G}}(k) \right\}_{k \in \mathbb{N}} = \left\{ \text{HYB}_{\mathsf{HS},\mathcal{B}_3,\mathcal{G}}(k) \right\}_{k \in \mathbb{N}} .$$

We can now focus on how to simulate for an adversary of the form $\mathcal{B}_3$.

The simulator $\mathcal{S}_3$ works as follows:

1. Initialize $\omega_0^{\mathsf{h}}$ as the preprocessing would have done.
2. Run $\mathcal{B}_3$ until it executed command (EXEC) exactly $\ell_K$ times. Simulate by doing noting.
3. Run $\mathcal{B}_3$ until it executed commands $(\texttt{REPLACE}, 0, I+i, x_i)$ for $i = 0, \ldots, \ell_I - 1$, in that sequence. Simulate the $i$-th replace by setting $\omega_0^{\mathsf{h}}[I + i] \leftarrow x_i$.
4. Let $x = \mathcal{X}^{-1}(x_0, \ldots, x_{\ell_I - 1})$ and query the ideal model to learn $y = \mathcal{G}_{\mathsf{K}}(x)$.
5. Execute command (EXEC) exactly $\ell_G$ times. Simulate by doing nothing.
6. Compute $(y_0, \ldots, y_{\ell_O - 1}) = \mathcal{Y}^{-1}((x_0, \ldots, x_{\ell_I - 1}), y)$.[20]
7. Let $i = 0$.
8. As long as $i < \ell_O$, run $\mathcal{B}_3$ to make it give the command (EXEC). In response to this update $\omega_0^{\mathsf{h}}[O + i] \leftarrow y_i$, and let $i \leftarrow i + 1$.

[19] Essentially we move to a non-adaptive adversary in order to ease the construction of the simulator which we describe next.

[20] Here we make use of the fact that the simulator $\mathcal{Y}^{-1}$ gets the tuple $(x_0, \ldots, x_{\ell_I - 1})$ as additional input. We emphasize that the simulator $\mathcal{S}_3$ also works in case $\mathcal{Y}^{-1}$ does not get $(x_0, \ldots, x_{\ell_I - 1})$ as input. However, since giving this tuple to $\mathcal{Y}^{-1}$ does *not* weaken the security definition, and moreover it makes the description of $\mathcal{S}_3$ simpler, we prefer to use the first formulation.

9. Run $\mathcal{B}_3$ until it executed the command (`EXEC`) once. Then restart from Step 2.

It follows from *Output Simulatability* that

$$\left\{\text{HYB}_{\mathsf{HS},\mathcal{B}_3,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} = \left\{\text{IDEAL}_{\mathcal{S}_3,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}}.$$

We can then construct the final simulator as follows, from $\mathcal{B}$, construct $\mathcal{B}_1$, from this $\mathcal{B}_1$ construct $\mathcal{B}_2$, from this $\mathcal{B}_2$ construct $\mathcal{B}_3$, and from this $\mathcal{B}_3$ get $\mathcal{S}_3$ and let $\mathcal{S} = \mathcal{S}_3$. It is a corollary to the above analysis that

$$\left\{\text{HYB}_{\mathsf{HS},\mathcal{B},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} =_{2^{-k}} \left\{\text{IDEAL}_{\mathcal{S},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}}.$$

This show that $\mathsf{HS}$ is RAM simulatable. Notice that we did not use anything assumptions on the size of the disk, so $\mathsf{HS}$ is also extra-space insensitive. Finally, notice that because $\mathsf{HS}$ self-destruct the first time it reads up a value from the secret domain which is not the correct value and because the correct values by *Constant Fan-Out* are touched at most $\alpha$ times, it follows that to prove that the scheme is $(\alpha+1)$-bounding, it is sufficient to prove that it is $(\alpha+1)$-bounding against an adversary of the form $\mathcal{B}_3$. In $\text{HYB}_{\mathsf{HS},\mathcal{B}_3,\mathcal{G}}(k)$ all values at input positions are written once and then read at most $\alpha$ times. Values at key positions and code positions are written once and read once, and $\alpha + 1 \geq 2$. Values at intermediary positions are written once and read at most $\alpha$ times.

This concludes the proof of Theorem 5.

# References


[1] Divesh Aggarwal, Yevgeniy Dodis, and Shachar Lovett. Non-malleable codes from additive combinatorics. *IACR Cryptology ePrint Archive*, 2013:201, 2013.

[2] Shashank Agrawal, Divya Gupta, Hemanta K. Maji, Omkant Pandey, and Manoj Prabhakaran. Explicit non-malleable codes resistant to permutations. *IACR Cryptology ePrint Archive*, 2014:316, 2014.

[3] Ross Anderson and Markus Kuhn. Tamper resistance: a cautionary note. In *WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[4] Mihir Bellare and David Cash. Pseudorandom functions and permutations provably secure against related-key attacks. In *CRYPTO*, pages 666–684, 2010.

[5] Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: Rka-prps, rka-prfs, and applications. In *EUROCRYPT*, pages 491–506, 2003.

[6] Mihir Bellare, Kenneth G. Paterson, and Susan Thomson. Rka security beyond the linear barrier: Ibe, encryption and signatures. In *ASIACRYPT*, pages 331–348, 2012.

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

[8] Mahdi Cheraghchi and Venkatesan Guruswami. Capacity of non-malleable codes. In *ITCS*, pages 155–168, 2014.

[9] Mahdi Cheraghchi and Venkatesan Guruswami. Non-malleable coding against bit-wise and split-state tampering. In *TCC*, pages 440–464, 2014.

[10] Sandro Coretti, Ueli Maurer, Björn Tackmann, and Daniele Venturi. From single-bit to multi-bit public-key encryption via non-malleable codes. *IACR Cryptology ePrint Archive*, 2014:324, 2014.

[11] Dana Dachman-Soled and Yael Tauman Kalai. Securing circuits against constant-rate tampering. In *CRYPTO*, pages 533–551, 2012.

[12] Dana Dachman-Soled and Yael Tauman Kalai. Securing circuits and protocols against 1/poly(k) tampering rate. In *TCC*, pages 540–565, 2014.

[13] Ivan Damgård, Sebastian Faust, Pratyay Mukherjee, and Daniele Venturi. Bounded tamper resilience: How to go beyond the algebraic barrier. In *ASIACRYPT (2)*, pages 140–160, 2013.

[14] Stefan Dziembowski and Sebastian Faust. Leakage-resilient circuits without computational assumptions. In *TCC*, pages 230–247, 2012.

[15] Stefan Dziembowski, Tomasz Kazana, and Maciej Obremski. Non-malleable codes from two-source extractors. In *CRYPTO (2)*, pages 239–257, 2013.

[16] Stefan Dziembowski, Krzysztof Pietrzak, and Daniel Wichs. Non-malleable codes. In *ICS*, pages 434–452, 2010.

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

[18] Sebastian Faust, Pratyay Mukherjee, Daniele Venturi, and Daniel Wichs. Efficient non-malleable codes and key-derivation for poly-size tampering circuits. In *EUROCRYPT*, pages 111–128, 2014.

[19] Sebastian Faust, Krzysztof Pietrzak, and Daniele Venturi. Tamper-proof circuits: How to trade leakage for tamper-resilience. In *ICALP (1)*, pages 391–402, 2011.

[20] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting circuits from leakage: the computationally-bounded and noisy cases. In *EUROCRYPT*, pages 135–156, 2010.

[21] Niklas Frykholm. Countermeasures against buffer over flow attacks. Technical report, RSA Data Security, Inc., November 2000.

[22] Shafi Goldwasser and Guy N. Rothblum. How to compute in the presence of leakage. In *FOCS*, pages 31–40, 2012.

[23] Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and David Wagner. Private circuits II: Keeping secrets in tamperable circuits. In *EUROCRYPT*, pages 308–327, 2006.

[24] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.

[25] Yael Tauman Kalai, Bhavana Kanukurthi, and Amit Sahai. Cryptography with tamperable and leaky memory. In *CRYPTO*, pages 373–390, 2011.

[26] Aggelos Kiayias and Yiannis Tselekounis. Tamper resilient circuits: The adversary at the gates. In *ASIACRYPT (2)*, pages 161–180, 2013.

[27] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.

[28] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[29] Feng-Hao Liu and Anna Lysyanskaya. Algorithmic tamper-proof security under probing attacks. In *SCN*, pages 106–120, 2010.

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

[31] Eric Miles and Emanuele Viola. Shielding circuits with groups. In *STOC*, pages 251–260, 2013.

[32] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):File 14, 1996.

[33] Martin Otto. *Fault Attacks and Countermeasures*. PhD thesis, University of Paderborn, Germany, 2006.

[34] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *EUROCRYPT*, pages 142–159, 2013.

[35] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *CHES*, pages 2–12, 2002.

[36] Hoeteck Wee. Public key encryption against related key attacks. In *Public Key Cryptography*, pages 262–279, 2012.

# A Proof of Theorem 1

**Theorem 1.** *Let* $\mathcal{C} = (\text{Init}, \text{Encode}, \text{Decode})$ *be a* $(\mathsf{lb}_{\mathsf{code}}, q)$*-CNMLR code, then* $\mathcal{C}$ *is also adaptively* $m$*-composable for any polynomial* $m = poly(k)$*.*

*Proof.* We assume that there exists a PPT adversary $\mathcal{A}$ such that:

$$\Pr[\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(1) = 1] - \Pr[\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(0) = 1] > \varepsilon \tag{1}$$

for some $\varepsilon$. The proof is by a hybrid argument where we replace in each hybrid game one of the loops of Definition 2 with a fixed choice of either $x^i_0$ or $x^i_1$. More precisely, in hybrid $i$, we append in the first $j \leq i$ iterations of the loop an encoding of $x^j_0$, while in the iterations $m \geq j > i$, we append an encoding of $x^j_1$. More formally, for any $j \in [m]$ we have:

$\underline{\text{GAME}^{j,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}}$

Compute $\Omega \leftarrow \text{Init}(1^k)$ and obtain $(x_0^1, x_1^1) \leftarrow \mathcal{A}(\Omega)$. Set $\mathbf{c}_0 = \emptyset, \mathbf{c}_1 = \emptyset$

For $i = 1, \ldots, m-j$ do the following:

Compute $(c_0^i, c_1^i) \leftarrow \text{Encode}(x_0^i)$ and update $(\mathbf{c}_0, \mathbf{c}_1)$ with $((\mathbf{c}_0, c_0^i), (\mathbf{c}_1, c_1^i))$.

Receive $(x_0^{i+1}, x_1^{i+1}) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$.

For $i = m-j+1, \ldots, m$ do the following:

Compute $(c_0^i, c_1^i) \leftarrow \text{Encode}(x_1^i)$ and update $(\mathbf{c}_0, \mathbf{c}_1)$ with $((\mathbf{c}_0, c_0^i), (\mathbf{c}_1, c_1^i))$.

Receive $(x_0^{i+1}, x_1^{i+1}) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$.

Receive $b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$.

Output $b'$.

Notice that $\text{GAME}^{0,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}} \equiv \text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(0)$ and $\text{GAME}^{m,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}} \equiv \text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(1)$. So,

$$\begin{aligned}
&\Pr[\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(1) = 1] - \Pr[\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(0) = 1] = \\
&\Pr[\text{GAME}^{m,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}} = 1] - \Pr[\text{GAME}^{0,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}} = 1] = \\
&\sum_{j=1}^{m} \Big( \Pr[\text{GAME}^{j,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}} = 1] - \Pr[\text{GAME}^{j-1,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}} = 1] \Big)
\end{aligned}$$

So, by Eq. (1)

$$\exists j \in [m] \ : \ \Pr[\text{GAME}^{j,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}} = 1] - \Pr[\text{GAME}^{j-1,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}} = 1] > \frac{\varepsilon}{m} \ .$$

Now we construct another PPT adversary $\mathcal{B}$ which is trying to distinguish between $\text{GAME}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(0)$ and $\text{GAME}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(1)$ (i.e., its challenge oracles) with black-box access to $\mathcal{A}$. We can assume without loss of generality that $\mathcal{A}$ does not violate the leakage bound. I.e., it never makes the leakage oracle return $\perp$. This frees us from keeping the leakage tallies. The reduction works as follows, where the description of simulation access to the leakage and tampering oracles is given below.

1. Receive $\Omega$ from the challenger and obtain $(x_0^1, x_1^1) \leftarrow \mathcal{A}(\Omega)$.
2. For $1 \le i \le m-j-1$ do the following:
   (a) Compute $(c_0^i, c_1^i) \leftarrow \text{Encode}(x_0^i)$ and update $(\mathbf{c}_0, \mathbf{c}_1)$ with $((\mathbf{c}_0, c_0^i), (\mathbf{c}_1, c_1^i))$.
   (b) Receive $(x_0^{i+1}, x_1^{i+1}) \leftarrow \mathcal{A}^{Sim-\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),Sim-\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),Sim-\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$
3. For $i = m-j$, proceed as follows:
   (a) Send $(x_0^i, x_1^i)$ to the challenger and update $(\mathbf{c}_0, \mathbf{c}_1)$ with $((\mathbf{c}_0, ?), (\mathbf{c}_1, ?))$. Notice that the challenger will produce an encoding $(c_0^i, c_1^i) \leftarrow \text{Encode}(x_b^i)$ and gives $\mathcal{B}$ access to it via its oracles. Notice that $(c_0^i, c_1^i)$ are only know through the challenge oracles.
   (b) Receive $(x_0^{i+1}, x_1^{i+1}) \leftarrow \mathcal{A}^{Sim-\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),Sim-\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),Sim-\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$.
4. For $i = m-j+1, \ldots, m$, proceed as follows:
   (a) Compute $(c_0^i, c_1^i) \leftarrow \text{Encode}(x_1^i)$ and update $(\mathbf{c}_0, \mathbf{c}_1)$ with $((\mathbf{c}_0, c_0^i), (\mathbf{c}_1, c_1^i))$.

(b) Receive $(x_0^{i+1}, x_1^{i+1}) \leftarrow \mathcal{A}^{Sim-\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),Sim-\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),Sim-\mathcal{O}^{q}_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$.

5. Receive $b'$ from $\mathcal{A}^{Sim-\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0,\cdot),Sim-\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1,\cdot),Sim-\mathcal{O}^{q}_{\mathsf{cnm}}((\mathbf{c}_0,\mathbf{c}_1),(\cdot,\cdot))}$.

6. Return $b'$.

We now describe how $\mathcal{B}$ simulates access to the oracles.

**Access to leakage oracle** $Sim - \mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_0, \cdot)$**:** For the first $m - j - 1$ rounds $\mathcal{B}$ has complete knowledge of $(\mathbf{c}_0, \mathbf{c}_1)$ and hence can easily simulate access to the oracle. For all rounds $\geq m-j$ on input a leakage query $(S, \mathsf{L})$, if $m - j \notin S$ (i.e., the adversary does not ask for leakage on the target encoding), then return $\mathsf{L}\{\mathbf{c}_0[i]\}_{i\in S}$. If $m - j \in S$, then hard-wire $\{\mathbf{c}_0[i]\}_{i\in S\setminus\{m-j\}}$ into the description of the leakage function $\mathsf{L}'(\mathtt{x})$ and submit it to $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_0^i, \cdot)$. Send the value returned from $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_0^i, \cdot)$ to $\mathcal{A}$.

**Access to leakage oracle** $Sim - \mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1, \cdot)$**:** This is simulated as in the previous step.

**Access to tampering oracle** $Sim - \mathcal{O}^{q}_{\mathsf{cnm}}((\mathbf{c}_0, \mathbf{c}_1), (\cdot, \cdot))$**:** For the first $m - j - 1$ rounds $\mathcal{B}$ has complete knowledge of $(\mathbf{c}_0, \mathbf{c}_1)$ and hence can easily simulate access to the oracle. For all rounds $\geq m - j$, $\mathcal{B}$ can simulate the tampering oracle $\mathcal{O}^{q}_{\mathsf{cnm}}((\mathbf{c}_0, \mathbf{c}_1), (\cdot, \cdot))$ as follows. Let $(\mathbf{c}_0, \mathbf{c}_1)$ be the vectors kept by $\mathcal{B}$ where $(\mathbf{c}_0, \mathbf{c}_1)[m-j] = (?, ?)$. On input $(\mathsf{T}_0, \mathsf{T}_1)$ that operate on vectors the adversary hard-wires $(\mathbf{c}_0, \mathbf{c}_1)[\ell]$ (for $\ell \neq m - j$) into $\mathsf{T}'_0$ and $\mathsf{T}'_1$ respectively. At position $m - j$ it will use the challenge encoding $(c_0^{m-j}, c_1^{m-j})$.

Next, $\mathcal{B}$ submits the such prepared functions $\mathsf{T}'_0$ and $\mathsf{T}'_1$ to its challenge oracle $\mathcal{O}^{q}_{\mathsf{cnm}}((c_0^{m-j}, c_1^{m-j}), (\mathsf{T}'_0, \mathsf{T}'_1))$. Let $c' \in \{0,1\}^{2n} \cup \{\perp, \mathsf{same}^\star\}$ be the value returned by the oracle. In case $c' = \mathsf{same}^\star$, return $(\mathsf{same}^\star, m - j)$ to $\mathcal{A}$. Else, in case $c' = (c'_0, c'_1)$ equals $(\mathbf{c}_0, \mathbf{c}_1)[\ell]$ (for some $\ell \neq m - j$), return $(\mathsf{same}^\star, \ell)$ to $\mathcal{A}$. Otherwise return $c'$.

Now it is easy to see that when the adversary $\mathcal{B}$ is in $\textsc{Game}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(0)$, it perfectly simulates $\textsc{Game}^{j-1,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}$ and when is in $\textsc{Game}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(1)$, it simulates $\textsc{Game}^{j,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}$. So,

$$\Pr[\textsc{Game}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(1) = 1] - \Pr[\textsc{Game}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(0) = 1] \geq \varepsilon/m \ .$$

Since $\mathcal{C}$ is a CNMLR code, we have that $\varepsilon/m$ is negligible, from which we get that $\varepsilon$ is negligible, as desired. □