An extended abstract of this paper is published in the proceedings of the 18th International Conference on Practice and Theory of Public-Key Cryptography—PKC 2015. This is the full version.

A Tamper and Leakage Resilient von Neumann Architecture

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

[2]*Department of Computer Science, Aarhus University*
[1]*Security and Cryptography Laboratory, EPFL*
[3]*Department of Computer Science, Sapienza University of Rome*

February 18, 2015

Abstract

We present a *universal framework* for tamper and leakage resilient computation on a von Neumann Random Access Architecture (RAM in short). The RAM has one CPU that accesses a storage, which we call the disk. The disk is subject to leakage and tampering. So is the bus connecting the CPU to the disk. We assume that the CPU is leakage and tamper-free. For a fixed value of the security parameter, the CPU has *constant size*. Therefore the code of the program to be executed is stored on the disk, i.e., we consider a von Neumann architecture. The most prominent consequence of this is that the code of the program executed will be subject to tampering.

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 disk along with the program to evaluate the primitive on that key. Our compiler only assumes the existence of a so-called continuous non-malleable code, and it only needs black-box access to such a code. No further (cryptographic) assumptions are needed. This in particular means that given an information theoretic code, the overall construction is information theoretic secure.

Although it is required that 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 yet universal component.

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.

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 [23, 22]. 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 behavior as $\mathcal{G}_{\mathsf{K}}$ but additionally remains secure in a non-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 K' in some adversarial way. Formally, security is typically modeled 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 [22, 19, 10, 25, 11]. That is, given a functionality $\mathcal{G}_{\mathsf{K}}$ that, e.g., computes the AES with key K, the compiler outputs a transformed functionality $\mathcal{G}'_{\mathsf{K}'}$ 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). Also, it is not allowed to re-wire the circuit.

The second approach is based on the 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 access 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, 26, 15, 1, 6, 7, 17, 18, 2, 9, 8]), 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 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 approach described above, which requires a specifically tailored tamper-proof hardware for each functionality that we intend to protect. Our solution is hence in spirit of earlier works (e.g., [19]) and reduces the problem of protecting arbitrary complicated computation to shielding a single, simple component.

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

We notice that while our construction works generically for any non-malleable code that satisfies certain composability properties (as explained in more detail below), we will focus in the following exposition mainly on non-malleable codes in the split-state setting. In this well-known setting (c.f. [26, 1, 15, 17, 7]) the codeword consists of two parts and the adversary is allowed to tamper independently with them in an arbitrary way.

1.1 Our Model

We put forward a generic model of a tamper and leakage resilient von Neumann random access architecture (alternatively called RAM architecture). To use the established terminology of leakage and tamper resilient compilers, we phrase the model in terms of computing keyed functionalities $\mathcal{G}_{\mathsf{K}}(\cdot)$. However, the model capture arbitrary poly-time computation which keeps a secret state that is initially K.

RAM schemes. We will use a *RAM scheme* to denote a RAM architecture $\mathbf{R}$ and a compiler $\mathbf{C}$ for $\mathbf{R}$. The RAM $\mathbf{R}$ has a disk D and a tamper/leakage-proof CPU that is connected with the disk through buses. The RAM compiler $\mathbf{C}$ takes as input the description of a functionality $\mathcal{G}$ and a key K and outputs an initial encoding of the disk. Inputs to the program are given by writing it

on the disk, and outputs are received by reading a special section of the disk. The program runs in *activations*. An activation denotes the time period of evaluating $\mathcal{G}_{\mathsf{K}}(\cdot)$ on some input x. An activation involves several *steps* of the CPU. In each step, the CPU loads a constant number of words from the disk (this might include reading part of the input), executes one computation on the loaded data, and writes the result back to the disk (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 steps. The CPU contains the following public untamperable components (i) a program counter `pc`, (ii) an activation counter `ac` and (iii) a self-destruct bit `B`. The activation counter `ac` is incremented after each activation, and the program counter `pc` specifies, during each activation, 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.

Security. We define security of a RAM scheme via the real-ideal simulation paradigm. In the real world the compiler $\mathbf{C}$ is run in order to produce the initial contents of the disk. As in previous works on tamper and leakage resilient compilers the pre-processing in the setup is assumed to be tamper and leakage proof and is executed once at the initialization of the system. Think of it as the setup running on a separate, possibly more secure machine. In the online phase, the adversary can specify between steps of the CPU a tampering function $\mathsf{Tamper}(\cdot)$ that modifies the disk: $D \leftarrow \mathsf{Tamper}(D)$. It can also specify a leakage function Leak and will then be given $\mathsf{Leak}(D)$. 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) disk. When requesting the next step it also specifies a leakage function $\mathsf{Leak}_{\mathsf{Bs}}$ and is given back $\mathsf{Leak}_{\mathsf{Bs}}(\mathsf{Bs})$, where Bs contains the values that were loaded or stored by the CPU.

Clearly, no computation is secure in the presence of arbitrary leakage and tampering. We therefore introduce a notion of *adversary class* to restrict the tampering and leakage queries that the adversary can submit. 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 RAM scheme is $\mathbf{A}$-secure if for all efficient adversaries from $\mathbf{A}$ 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.

We also introduce a notion of secure emulation. An emulator takes as input a RAM scheme (think of a RAM scheme for an idealised highly secure RAM) and outputs another RAM scheme (think of a RAM scheme for more real-world-like highly insecure RAM). We define the notion of security of an emulator such that if one is given a secure RAM scheme for the idealised RAM and applies a secure emulator, then one gets a secure RAM scheme for the less secure architecture. This allows to do modular proofs.

1.2 Motivation and Challenges of our Model

On RAM computation vs. circuits. The reasons why we want to lift the study of leakage and tamper resilience to the RAM setting are motivated by practice. It is well known that computing a function using a circuit instead of a RAM can yield a quadratic blow-up in complexity. Even worse, in a setting as ours, where the data (the encoding of K) is already laid out, the complexity can suffer an exponential blow-up, if a given activation only reads a small part of the key. Furthermore,

it seems a simpler task in practice to produce a lot of tamper proof copies of a small universal piece of hardware than to produce different tamper proof circuits for different desired functionalities.

On the trusted CPU assumption. As non-malleable codes typically do not have any homomorphic properties that enable computation,[1] we assume a tamper and leakage-proof CPU that carries out decoding. The CPU is the only part of the computation that is completely trusted. Notice that 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 disk, decodes them, executes some instruction (that can be as simple as a NAND operation) and writes the encoded result back to the disk. Notice that in contrast to earlier work on tamper resilient compilers based on non-malleable codes [16, 26, 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, 26, 17].

One may object that given such a powerful tamper-proof component a solution for tamper and leakage resilience is simple. Let us take a look at an adversary that can apply powerful tampering functions to the state of the disk between executions of the CPU. To this end, observe that 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.[2] The adversary can also 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.

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. Section 1.4 for more details on this).

On the counters. In our model the CPU has public untamperable counters. The reason is that in order to tolerate leakage from the buses (connecting the CPU and the disk), we must make sure that the state of the CPU changes after each step. Otherwise, one may execute the following "reset-and-leak attack". The tampering functions can reset the disk to previous states an *unbounded* number of times, and without the counters, the CPU is also always in the same state at the start of an execution, so it would read the same values repeatedly. Notice that, as we allow leakage from the buses, each time the CPU loads a value it leaks through the bus. So, loading any value repeatedly an unbounded number of times implies that all the values on the disk could eventually be leaked at some point. We also stress that we pick a public value for this purpose and not a secret register as

[1] In fact, a homomorphism would in many cases contradict the non-malleability property of the code.

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

we want to minimize the assumption on the hardware—and of course secret un-tamperable memory is a much stronger assumption than public un-tamperable memory.

Moreover, assuming only counters makes our model a *strict generalization* of the circuit model: we can make an equivalent circuit where each gate can be thought of as one invocation of the CPU. Each gate will be identical to the CPU, except that it has the appropriate counters hard-coded into it. Assuming secret registers would not make such a transformation to circuitry possible.

On the self-destruct bit. In addition to the counter we use a *tamper-proof* "self-destruct" bit in our construction. Firstly, such bit is used to serve the same purpose as in the tamper-resilient compiler of [17]: it acts as a flag indicating that tampering has been detected for the first time and, if the execution does not stop at this point, the adversary can continue to learn information on the codeword (eventually recovering the whole codeword) which should, of course, be prevented.[3] Moreover, one may notice that without having a self-destruct bit, it is impossible to tolerate leakage from the buses. Consider, again, the "reset-and-leak attack" described above. The untamperable program counter enables the CPU to detect that a "reset" has taken place (i.e., values read from the disk do not match its internal state). However, at this point it is too late: the values were already on the buses, and hence subject to leakage. In this case the self-destruct bit allows the CPU to stop execution the first time such tampering is detected.

We also stress that having one bit, which is in fact "one-time writable", is optimal. Moreover, this seems as a reasonable hardware assumption: one can think of the CPU having a fuse that it can blow once (and check if it was ever blown).

On minimizing hardware assumptions. We emphasize that the main goal of this work is to study feasibility to securely execute *any* computation in the presence of very strong leakage and tampering attacks (in particular we consider *arbitrary* continuous leakage from buses and *arbitrary* tampering in the split-state model). We show that indeed this can be achieved by a simple, universal, constant-size CPU that is fully trusted. The CPU does not keep any secret state, and only has a short public un-tamperable memory that keeps the program counter (of size logarithmic in the security parameter) and the self-destruct bit. We notice that one can develop easier solutions if the CPU can keep a large, mutable, secret state between executions. In this case the CPU could encrypt the disk and authenticate it using, e.g., a Merkle tree. Of course, keeping a secret state between executions of the CPU is a much stronger hardware assumption.

1.3 Our Techniques

We construct our RAM scheme in two steps. We first formulate a hybrid model, which is a wishful RAM architecture where there is no leakage from the disk, no leakage from the bus and where the only allowed tampering is of the following types: (i) the adversary might copy a word from one position of the disk to another position on the disk (without looking at the value), and (ii) he might overwrite a position on the disk with a word of an arbitrary choice. As a first step we show how to compile securely to this hybrid platform. We then show how to use a non-malleable code to emulate this platform. Below we first describe the compiler, and then the emulator.

[3]For example, the tampering function can make the codeword "valid" or "invalid" depending on the first bit of the codeword, and hence learn the first bit based on the outcome.

The compiler. We construct a RAM scheme for the hybrid architecture described above. We need to mitigate the fact that the adversary can overwrite values and copy them around. 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 6) 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 our hybrid compiler so-called c-bounded, as required by the emulator (see below).

Notice that this compiler uses no cryptography, so it is information-theoretic secure. Hence, if we can emulate the hybrid architecture with information-theoretic security, the overall security will be information theoretic!

The emulator. The basic idea of the emulator is simple. Given a RAM scheme for the hybrid model and a non-malleable code, each value of the disk is encoded using the code. The CPU will then decode the values after loading them, compute as the CPU of the hybrid scheme and then encode the results and put them back on disk. Intuitively, a non-malleable code has the property that if a codeword is changed it either becomes invalid or an encoding of an unrelated value (known by the adversary). Since codewords can of course be copied around without modifying them, it seems intuitive that the above emulator should work if the RAM only allows leakage and tampering that the code is designed to tolerate. We can in fact take this as an informal definition and say that a given non-malleable code *fits* a given RAM architecture (given by the CPU and the adversary class) if for all hybrid schemes the natural emulator sketched above securely emulates the hybrid scheme. With this definition, we tautologically get that if there is a non-malleable code fitting a given RAM architecture, then there is also a secure RAM scheme for that architecture, namely apply the natural emulator to our secure compiler from above.

We exemplify our approach by showing that the split-state continuous non-malleable code (CNMC) from [17] fits a split-state RAM, where the disk is split into two disks and the adversary is allowed arbitrary independent tampering of each disk. In contrast to traditional non-malleable codes, *continuous* non-malleability guarantees that the code remains secure under continuous attacks without assuming erasures. The natural emulator uses many encodings, so the 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.[4]

We remark that the code construction of [17] is in the common reference string (CRS) model, meaning that at setup a public string `crs` is generated and made available to all parties. Importantly, the security of the code requires that the adversary is not allowed to modify `crs`. Similarly, when one uses the code of [17] within our framework, the CRS is assumed to be un-tamperable and chosen by a trusted party; for instance, it can be chosen at production time and be hard-coded into the CPU of the RAM. However, the CRS can be public, and in particular the tampering and leakage from the disks can fully depend on it. Also the CRS is generated once and for all, so it perfectly matches our assumption of having a universal component (the CPU) that can be used to protect arbitrary computation. The assumption of having a public un-tamperable CRS is not new; see, e.g., [24, 26] for further discussion.

Bounding RAM 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 reveal the entire code. Our compiler is constructed to have this property. 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.

1.4 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 [4, 3, 24, 29, 5, 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 [23, 20, 14, 21, 28, 27]. While most works on leakage resilient circuit compilers require leakage-proof hardware, the breakthrough work of Goldwasser and Rothblum [21] 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 [21] 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.

The concept of tamper resilient circuits has been introduced by Ishai, Prabhakaran, Sahai and Wagner [22] and further studied in [22, 19, 10, 25, 11]. On the upside such compilers require simpler

[4] In [9] Coretti *et al.* show that the information theoretic construction of [16] in the bit-wise tampering (and no leakage) model is continuously non-malleable, so in that setting our compiler would be information theoretic, albeit only protecting against a weaker adversary class.

tamper-proof hardware,[5] but study a weaker tampering model. Concretely, they assume that an adversary can tamper with individual wires (or constant size gates [25]) 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 as the program code is read from the disk. Translating this to a circuit model would essentially allow the adversary to "re-wire" the circuit.

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 on encodings.

Concurrent and independent work. A concurrent and independent paper [12] gives a related result on protecting RAM schemes against memory leakage and tampering. The main difference with the setting considered in this paper is that their model does not cover "reset attacks", i.e., the tampering functions are not allowed to keep a backup storage where previous codewords are stored and continuously tampered. This is enforced in their construction by assuming perfect erasures.

Technically the solutions are very different. Instead of encoding each element on the disk via a non-malleable code, the scheme of [12] encodes only the registers of the CPU to virtually equip it with secret registers, and then uses disk encryption to secure the disk; this can be phrased as using a non-malleable code with local properties. Finally, the scheme of [12] incorporates directly an ORAM, whereas we propose to view this as a separate step. First applying an ORAM and then our compiler will yield a scheme with the same asymptotic complexity of the one in [12]. However, as long as non-malleable codes are less efficient in practice than symmetric encryption, the scheme of [12] appears more practical. On the other hand, if we base our construction on an information theoretically secure code, the whole construction has unconditionally security. The solution in [12] is inherently computational.

2 Preliminaries

2.1 Notation

For $n \in \mathbb{N}$, we write $[n] := \{1, \ldots, n\}$. Given a set $\mathcal{X}$, we write $x \leftarrow \mathcal{X}$ to denote that element x is sampled uniformly from $\mathcal{X}$. If A is an algorithm, $y \leftarrow \mathsf{A}(x)$ denotes an execution of A with input x and output y; if A 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 $X_1 \approx_{\mathrm{c}} X_2$ to denote that X_1 and X_2 are computationally indistinguishable meaning that for all PPT algorithms $\mathcal{A}$ we have that $\Pr[\mathcal{A}(X_1) = 1] - \Pr[\mathcal{A}(X_2) = 1] \leq negl(k)$.

[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 [22, 19] it is allowed that faults are persistent so at some point the entire circuitry may be subject to tampering.

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 $\mathcal{C} = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Decode})$, is a triple of algorithms specified as follows: (1) Init, takes as input the security parameter and outputs a public common reference string $\mathtt{crs} \leftarrow \mathsf{Init}(1^k)$; (2) Encode, takes as input a string $x \in \{0,1\}^\ell$, for some fixed integer ℓ, and the public parameters, and outputs a codeword $c = (c_0, c_1) \leftarrow \mathsf{Encode}(\mathtt{crs}, x)$ 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 $x = \mathsf{Decode}(\mathtt{crs}, c)$ where $x \in \{0,1\}^\ell \cup \{\perp\}$. We require that $\mathsf{Decode}(\mathtt{crs}, \mathsf{Encode}(\mathtt{crs}, x)) = x$ for all $x \in \{0,1\}^\ell$ and for all $\mathtt{crs} \leftarrow \mathsf{Init}(1^k)$. 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 $(c_0, c_1) \leftarrow \mathsf{Encode}(\mathtt{crs}, x_b)$, and (ii) the adversary obtains $\mathsf{Decode}^*(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1))$, where Decode^* is as Decode except that it returns a special symbol $\mathsf{same}^\star$ if $(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1)) = (c_0, c_1)$.

Next, we recall the notion of continuous non-malleable and leakage resilient codes as introduced in [17]. Intuitively, a code $\mathcal{C}$ is continuously non-malleable if even after continuous tampering with the the two halves of the codeword the adversary is not able to maul it to an encoding of a related value. This notion is formalized by the oracle $\mathcal{O}^q_{\mathsf{cnm}}((c_0, c_1), (\cdot, \cdot))$, which is parametrized by an encoding $(c_0, c_1) \leftarrow \mathsf{Encode}(\mathtt{crs}, x)$ for some value $x \in \{0,1\}^*$ and takes as input functions $\mathsf{T}_0, \mathsf{T}_1 : \{0,1\}^n \to \{0,1\}^n$, where $n = |c_i|$.

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

By "self-destruct" we mean that once $\mathsf{Decode}(\mathtt{crs}, (c'_0, c'_1))$ outputs $\perp$, the oracle will answer $\perp$ to any further query. The oracle is superscripted with q which denotes that in one experiment an oracle can be queried for at most q-times.

In the definition below, we use the leakage oracle $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_b, \cdot)$ (where $b \in \{0,1\}$) that can be queried on leakage function $\mathsf{L} : \{0,1\}^n \to \{0,1\}^\lambda$ such that $\lambda \leq \mathsf{lb}_{\mathsf{code}}$ and returns $\mathsf{L}(c_b)$.[7]

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

$\underline{\textsc{Game}^{\mathsf{cnmlr},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}(b)}$
Compute $\mathtt{crs} \leftarrow \mathsf{Init}(1^k)$ and give it to $\mathcal{A}$.
Receive (x_0, x_1) from $\mathcal{A}$ with $|x_0| = |x_1|$.
Compute $(c_0, c_1) \leftarrow \mathsf{Encode}(x_b)$.
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:

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

[7] Notice that our actual definition is stronger and lets the adversary submits multiple functions to the leakage oracles as long as the total leakage obtained from one oracle is smaller than $\mathsf{lb}_{\mathsf{code}}$.

We remark that depending on the actual code the public parameters $\mathtt{crs}$ can be empty. However, whenever present, they are assumed to be untamperable.[8]

We extend the notion of Definition 1 to a setting where there is a set of encodings rather than only one encoding and the adversary can adaptively leak and tamper on the encodings jointly. We will overload notation for ease of description—in particular, we will use the same notation for an oracle parametrized by a single encoding or a vector of encodings.

Consider a vector $\mathbf{x} = (x_1, \ldots, x_m) \in (\{0,1\}^*)^m$ 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 $(c_0^i, c_1^i) = \mathsf{Encode}(\mathtt{crs}, x_i)$ (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 $\mathsf{T}_0, \mathsf{T}_1 : \{0,1\}^{mn} \to \{0,1\}^n$.[9]

$\underline{\mathcal{O}^q_{\mathsf{cnm}}((\mathbf{c}_0, \mathbf{c}_1), (\mathsf{T}_0, \mathsf{T}_1))}$:
$(c_0', c_1') = (\mathsf{T}_0(\mathbf{c}_0), \mathsf{T}_1(\mathbf{c}_1))$
If $\exists\ i \in [m]$ such that $(c_0', c_1') = (c_0^i, c_1^i)$ return $(\mathsf{same}^\star, i)$
If $\mathsf{Decode}(\mathtt{crs}, (c_0', c_1')) = \perp$, 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 λ. It keeps a state what is the current amount of information $(\lambda_1, \ldots, \lambda_m)$ that has been leaked. Initially, we set $(\lambda_1, \ldots, \lambda_m) = (0, \ldots, 0)$.

$\underline{\mathcal{O}^{\mathsf{lb}}_{\mathsf{code}}(\mathbf{c}, (S, \mathsf{L}))}$:
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)**.** *Let* $\mathcal{C} = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Decode})$ *be a* $(\mathsf{lb}_{\mathsf{code}}, q)$*-CNMLR encoding scheme. For some adversary* $\mathcal{A}$ *consider the following interactive game:*

[8] This corresponds to the assumption that the common reference string cannot be modified in the construction of [17].

[9] 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.

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

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

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

Compute $(c_0^i, c_1^i) \leftarrow \mathsf{Encode}(\mathtt{crs}, x_b^i)$ 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:

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

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 $\{0,1\}^{(i-1)n}$.

In the following, we show that the Definition 2 and Definition 1 are equivalent (asymptotically). Clearly, adaptive composability implies continuous non-malleability for $m = 1$. The other direction follows by the following theorem.

Theorem 1. *Let $\mathcal{C} = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Decode})$ 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 ε. 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_0^i or x_1^i. More precisely, in hybrid i, we append in the first $j \leq i$ iterations of the loop an encoding of x_0^j, while in the iterations $m \geq j > i$, we append an encoding of x_1^j. More formally, for any $j \in [m]$ we have:

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

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

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

Compute $(c_0^i, c_1^i) \leftarrow \mathsf{Encode}(x_0^i)$ 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 $(c_0^i, c_1^i) \leftarrow \mathsf{Encode}(x_1^i)$ 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)

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

Now we construct another PPT adversary $\mathcal{B}$ which is trying to distinguish between $\textsc{Game}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(0)$ and $\textsc{Game}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(1)$ (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 $\mathtt{crs}$ from the challenger and obtain $(x_0^1, x_1^1) \leftarrow \mathcal{A}(\mathtt{crs})$.
2. For $1 \le i \le m - j - 1$ do the following:
   (a) Compute $(c_0^i, c_1^i) \leftarrow \mathsf{Encode}(x_0^i)$ 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 $(c_0^i, c_1^i) \leftarrow \mathsf{Encode}(x_b^i)$ 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 $(c_0^i, c_1^i) \leftarrow \mathsf{Encode}(x_1^i)$ 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 $\ge m-j$ on input a leakage query (S, 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 $\ge m - j$, $\mathcal{B}$ 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 $\ell \ne m - j$) into T'_0 and 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 T'_0 and 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 $\ell \ne m - j$), 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 $\text{GAME}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(0)$, it perfectly simulates $\text{GAME}^{j-1,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}$ and when is in $\text{GAME}^{\mathsf{cnmlr}}_{\mathcal{C},\mathcal{B}}(1)$, it simulates $\text{GAME}^{j,q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{A}}$. So,

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

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

3 A Generic Leakage and Tamper Resilient RAM

In this section we describe our model of a generic random access machine (RAM) architecture with a leakage and tamper resilient CPU and with memory and buses, which are subject to leakage. Our RAM architecture is meant to implement some keyed functionality $\mathcal{G}_{\mathsf{K}}$, e.g., an AES running with key K taking as input messages and producing the corresponding ciphertexts, but the model also applies to more general computations. The RAM has one tamperable and leaky disk D, and one CPU, which has a size independent of the function to be computed. We interchangeably denote the memory used by the CPU by "disk", "storage" and "memory"; this might physically be any kind of storage that the CPU can access. We assume there is a leak-free and tamper-free pre-processing phase, which outputs an encoding of the functionality $\mathcal{G}_{\mathsf{K}}$. One can think of this as a separate phase where a compiler is run, possibly on a different, more secure machine.

The initial encoding consists of data and instructions, which we store on the 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 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 disk allows to efficiently determine the input and output (in case the disk was not tampered). In the online phase, the CPU loads an instruction and data from the disk (as specified by the instruction). Reading from the disk might involve reading part of the input. Then it computes and stores back the intermediate results on the disk, and processes the next instruction. The next instruction is found on the disk at the location given by a program counter $\mathtt{pc}$, which is incremented by one in each invocation of the CPU and which is reset when the CPU raises a flag $\mathtt{T} = 1$. Writing to the disk could involve writing part of the output. The adversary is allowed to tamper and to leak from the disk between each two invocations of the CPU; furthermore the adversary is allowed to leak from the bus carrying the information between the CPU and the disk. In the following, we give a formal presentation of our model.

Specification of RAM. We use parameters $w, \tau, d, k \in \mathbb{N}$ below, where w is the word length, τ is length of an instruction type, d specifies the number of arguments of an instruction, and k is the security parameter. We require $w \geq \tau + 2kd$. We let the *disk* D be of length 2^k. This is just a convenient convention to avoid specifying a fixed polynomial-size disk. A poly-time program will access only polynomially many positions in the disk and all positions not yet written are by convention 0^w, so a disk D can at any time be represented by a poly-sized data structure. When we pass disks around in the below description, we mean that we pass such a poly-sized representation. We index a disk with $i \in [2^k]$. We also index the disk with bit-strings $i \in \{0,1\}^*$, by considering them binary numbers and then taking the result $\bmod 2^k$. An (τ, d)-bounded *instruction* $\mathcal{I}$ is defined as a quadruple $(\mathsf{Y}, \mathsf{I}, \mathsf{O}, \mathsf{Aux})$ where, $\mathsf{Y} \in \{0,1\}^\tau$, $\mathsf{I}, \mathsf{O} \in [2^k]^d$ and $\mathsf{Aux} \in \{0,1\}^{w-(\tau+2kd)}$. One may

think of Y as the type of operation (e.g., a NAND operation) that is computed by the instruction. The d-tuples I, O define the position on the disk where to read the inputs and where to write the outputs of the instruction. The string Aux is just auxiliary information used to pad to the right length. When we do not write it explicitly we assume it is all-0.

Formally, a RAM $\mathbf{R}$ is specified by $\mathbf{R} = (w, \tau, d, \mathsf{Init}, \mathsf{Random}, \mathsf{Compute})$ and consists of:

1. A disk $D \in (\{0,1\}^w)^{2^k}$.
2. Init: An algorithm that takes as input the security parameter 1^k, and returns a public common reference string $\mathtt{crs} \leftarrow \mathsf{Init}(1^k)$ (to be hard-coded into the CPU).
3. CPU: A procedure which is formally written as pseudo-code in Fig. 1. The CPU is connected to the disk by a bus Bs, which is 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 w bits. CPU has the public parameters $\mathtt{crs}$ hard-coded, and takes as inputs data sent through the bus, a strictly increasing activation[10] counter $\mathtt{ac}$, and a program counter $\mathtt{pc}$ which is strictly increasing within one activation and reset between activations. The CPU runs in three steps: (i) d loads, (ii) 1 computation and (iii) d stores. In the computation step CPU calls Random and $\mathsf{Compute}$ to generate fresh randomness and evaluate the instruction.
   (a) 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 CPU needs to stop execution, and the completion indicator bit $\mathtt{T}$ which indicates the completion of the current activation.

   CPU outputs the possibly updated disk D, the self-destruct indicator ($\mathtt{B}$) and the completion indicator ($\mathtt{T}$). Notice that the CPU does not need to take $\mathtt{B}$ 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 returns $\mathtt{T} = 1$ we reset $\mathtt{pc} = 0$ and increment the activation counter $\mathtt{ac}$. When the CPU returns $\mathtt{B} = 1$, the CPU self-destructs. After this no more execution of the CPU takes place.

Input and output to the program will be specified via the user/adversary reading and writing the disk. We therefore need a section of the disk that can be read and written at will. We call this the *public section*. We will model this by given the adversary full read/write access to $D_{\mathsf{pub}} = D[0, 2^{k-1} - 1]$ and limited access to $D_{\mathsf{sec}} = D[2^{k-1}, 2^k - 1]$. We call D_{pub} the public disk and we call D_{sec} the secret disk. Note that $D = D_{\mathsf{pub}} \| D_{\mathsf{sec}}$. Also note that the CPU is taking instructions from the public disk; this means that protecting the access pattern of the program has to be done explicitly.

RAM schemes. Informally, a RAM compiler $\mathbf{C}$ takes as input the description of a functionality $\mathcal{G}$ with secret key K, and outputs an encoding of the functionality itself, to be executed on a RAM $\mathbf{R}$. Formally, a *RAM compiler* $\mathbf{C}$ for $\mathbf{R}$ is a PPT algorithm which takes a keyed-function description

[10] 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 CPU is run once, one execution.

Input: $(\mathtt{crs}, D, \mathtt{pc}, \mathtt{ac}, \mathsf{Leak}_{\mathsf{Bs}})$
// Loading...
Parse $D[\mathtt{pc}]$ as an instruction $(\mathsf{Y}, \mathsf{I}, \mathsf{O}, \mathsf{Aux})$
Load $\mathtt{R}_0 \leftarrow (\mathsf{Y}, \mathsf{I}, \mathsf{O}, \mathsf{Aux})$
Initialize the bus $\mathsf{Bs} = (\mathtt{pc}, \mathtt{R}_0)$
for $j = 1 \to d$ **do**
  Let $\mathtt{loc}_j = \mathsf{I}[j]$ *// Load input from disk at position* $\mathsf{I}[j]$
  Load $\mathtt{R}_j \leftarrow D[\mathtt{loc}_j]$
  Set $\mathsf{Bs} \leftarrow (\mathsf{Bs}, \mathtt{loc}_j, \mathtt{R}_j)$ *// Write data from disk to bus*
end for
// Computing...
Sample $r \leftarrow \mathsf{Random}$
Compute $((\mathtt{O}_1, \ldots, \mathtt{O}_d), \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{Compute}(\mathtt{crs}, (\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_d), r, \mathtt{pc}, \mathtt{ac})$
// Storing...
for $j = 1 \to d$ **do**
  Let $\mathtt{loc}_j = \mathsf{O}[j]$
  Store $D[\mathtt{loc}_j] \leftarrow \mathtt{O}_j$ *// Store output on disk at position* $\mathtt{loc}_j$
  Set $\mathsf{Bs} \leftarrow (\mathsf{Bs}, \mathtt{loc}_j, \mathtt{O}_j)$
end for
Let $\lambda_{\mathsf{Bs}} = \mathsf{Leak}_{\mathsf{Bs}}(\mathsf{Bs})$ *// Compute leakage from the bus*
Output: $(D, \mathtt{B}, \mathtt{T}, \lambda_{\mathsf{Bs}})$

Figure 1: Algorithm CPU

$\mathcal{G}$ and a key $\mathsf{K} \in \{0,1\}^*$ as input, and outputs an encoding of the form $((\ell_P, I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}), \omega)$, called the *program*. Here $\omega = (\omega_{\mathsf{pub}}, \omega_{\mathsf{sec}})$ such that $\omega_{\mathsf{pub}}, \omega_{\mathsf{sec}} \in (\{0,1\}^w)^\ell$ for $\ell \leq 2^{k-1}$. When we say that we *store* ω on the disk we mean that we pad both of $\omega_{\mathsf{pub}}, \omega_{\mathsf{sec}}$ with 0s until they have length 2^{k-1}, giving values $\omega'_{\mathsf{pub}}, \omega'_{\mathsf{sec}}$ and then we assign $\omega'_{\mathsf{pub}} \| \omega'_{\mathsf{sec}}$ to D. We write ℓ_P for the program length, $I \geq \ell_P$ for the position where the input will be put on the disk, ℓ_I for the length of the input, $O \geq I + \ell_I$ for the position where the output is put on the disk, and ℓ_O for the length of the output such that $O + \ell_O \leq 2^{k-1}$. We think of the positions 0 to $\ell_P - 1$ as consisting of instructions, but make no formal requirement. 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 w (resp., as a value in the range of $\mathcal{G}_{\mathsf{K}}$).

We introduce a class $\mathbb{G}$ of functionalities $\mathcal{G}$ that a compiler is supposed to be secure for (e.g., all poly-time functionalities) and a class $\mathbb{P}$ of programs that a compiler is supposed to compile to (e.g., all poly-time programs). We use $\mathbf{C} : \mathbb{G} \to \mathbb{P}$ to denote that on input $\mathcal{G} \in \mathbb{G}$, the compiler $\mathbf{C}$ outputs a program in $\mathbb{P}$.

We define a *RAM scheme* RS as the ordered pair $(\mathbf{C}, \mathbf{R})$ such that $\mathbf{R}$ is a RAM and $\mathbf{C}$ a compiler for $\mathbf{R}$. The correctness of a RAM scheme is formalized via a game where we compare the execution of the RAM with the output of the original functionality $\mathcal{G}_{\mathsf{K}}$, upon an arbitrary sequence of inputs $(x_1, \ldots, x_N)$. 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 distributions except with negligible probability. This is formalized below.

Definition 3 (Correctness of a RAM Scheme)**.** *We say a RAM scheme* RS *is correct (for function class* $\mathbb{G}$ *and program class* $\mathbb{P}$*) if* $\mathsf{RS}.\mathbf{C} : \mathbb{G} \to \mathbb{P}$*, and for any function* $\mathcal{G} \in \mathbb{G}$*, 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:*

- *Sample* $\mathtt{crs} \leftarrow \mathbf{R}.\mathsf{Init}(1^k)$.
- *Run the compiler* $\mathbf{C}$ *on* $\mathtt{crs}, (\mathcal{G}, \mathsf{K})$ *to generate the encoding* $((I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}), \omega) \leftarrow \mathbf{C}(\mathtt{crs}, (\mathcal{G}, \mathsf{K}))$*, and store it into the disk of* $\mathbf{R}$ *as in* $D \leftarrow \omega$.
- *For* $i = 1 \to N$ *proceed as follows. Encode the input* $(x_{i,0}, \ldots, x_{i,\ell_I - 1}) \leftarrow \mathcal{X}(x_i)$*, store it on the disk* $D[I + j] \leftarrow x_{i,j}$ *(for* $0 \leq j < \ell_I$*) and run the following activation loop:*
  1. *Let* $\mathtt{ac} \leftarrow i$ *and* $\mathtt{pc} \leftarrow 0$.
  2. *Run* CPU *and update the disk* $(D, \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{CPU}(\mathtt{crs}, D, \mathtt{pc}, \mathtt{ac})$.[11]
  3. *If* $\mathtt{B} = 1$ *return* 0 *and halt.*
  4. *If* $\mathtt{T} = 0$*, then increment the program counter* $\mathtt{pc} \leftarrow \mathtt{pc} + 1$ *and go to Step 2. If* $\mathtt{T} = 1$*, let* $y_i \leftarrow \mathcal{Y}(D[O], \ldots, D[O + \ell_O - 1])$*. If* $y_i \neq \mathcal{G}_{\mathsf{K}}(x_i)$*, then return* 0 *and halt.*
- *Return* 1.

Security. We now proceed to define security of a RAM scheme, using the real-ideal paradigm. In the following we let k denote the security parameter. Consider a RAM scheme $\mathsf{RS} = (\mathbf{C}, \mathbf{R})$. First we run $\mathbf{C}$, which takes the description of $\mathcal{G}$ and a key K as inputs and generates an encoding of the form $((I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}), \omega)$. Then we store ω on the disk D and we advance to the online phase where the adversary $\mathcal{A}$ can run $\mathbf{R}$ on inputs of his choice. Formally, he is allowed to arbitrarily read from and write to D_{pub} and therefore also $D[I], \ldots, D[I + \ell_I - 1]$ and $D[O], \ldots, D[O + \ell_O - 1]$. Moreover, $\mathcal{A}$ can tamper with the secret disk D between each execution of the CPU. He specifies a function Tamper and the effect is that the disk is changes to $D \leftarrow \mathsf{Tamper}(D)$. The adversary can also leak from the disk between executions. He specifies a function Leak and he is given $\mathsf{Leak}(D)$. The adversary also decides when the CPU is invoked, and it gets to specify a leakage function $\mathsf{Leak}_{\mathsf{Bs}}$ for each invocation obtaining λ_{Bs} as defined in Fig.1. Besides the leakage from the bus, the procedure CPU is leakage and tamper proof.

We introduce the notion of an *adversary class*. This is just a subset $\mathbf{A}$ of all adversaries. As an example, $\mathbf{A}$ might be the set of $\mathcal{A}$ which leak at most 42 bits in total from the disk and which does the tampering in a split-state manner (more about this in the following).

We write $\textsc{Real}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)$ for the output distribution in the real execution and we let $\textsc{Real}_{\mathsf{RS},\mathcal{A},\mathcal{G}} = \{\textsc{Real}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)\}_{k \in \mathbb{N}}$. For a formal description see Fig. 2. A few remarks to the description are in order.

- **Adaptivity.** We stress that by writing the disk, the adversary is allowed to query the RAM on adaptively chosen inputs. Also note that the adversary can always hard-wire known values into a tampering command (e.g., values that were already leaked from the disk), and specify a tampering function that changes the content of the disk depending on the hard-wired values.
- **Tampering within executions.** Notice that the adversary is not allowed to tamper between two executions of the CPU. This is without loss of generality, as later we will allow the adversary to know the exact sequence of locations to be read by the CPU and hence, equivalently, the adversary can just load some location, tamper and then execute before loading

[11] When we do not specify a leakage function, we assume that it is the constant function outputting the empty string, and we ignore the leakage in the output vector.

1. Initialization: Sample $\mathtt{crs} \leftarrow \mathbf{R}.\mathsf{Init}(1^k)$. Sample the key 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 $\mathtt{B} \leftarrow 0$, and the activation indicator $\mathtt{T} \leftarrow 0$.
2. Pre-processing: Sample an encoding by running the compiler $(P, \omega_{\mathsf{pub}}, \omega_{\mathsf{sec}}) \leftarrow \mathbf{C}(\mathtt{crs}, (\mathcal{G}, \mathsf{K}))$, where $P = (I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y})$. Store the encoding $\omega = (\omega_{\mathsf{pub}}, \omega_{\mathsf{sec}})$ into the disk D. Give $(\mathtt{crs}, P, \omega_{\mathsf{pub}})$ 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 $\mathtt{CMD} = (\mathtt{LEAK}, \mathsf{Leak})$, compute $\lambda \leftarrow \mathsf{Leak}(D)$ and give λ to $\mathcal{A}$.
   (c) If $\mathtt{CMD} = (\mathtt{TAMPER}, \mathsf{Tamper})$ then modify D using the tampering function: $D \leftarrow \mathsf{Tamper}(D)$.
   (d) If $\mathtt{CMD} = (\mathtt{EXEC}, \mathsf{Leak}, D')$ and $\mathtt{B} = 0$ then proceed as follows:
      i. Update the public disk $D_{\mathsf{pub}} \leftarrow D'$.
      ii. Run CPU and update the disk: $(D, \mathtt{B}, \mathtt{T}, \lambda_{\mathsf{Bs}}) \leftarrow \mathsf{CPU}(\mathtt{crs}, D, \mathtt{pc}, \mathtt{ac}, \mathsf{Leak})$.
      iii. Give $(\mathtt{T}, \lambda_{\mathsf{Bs}}, D_{\mathsf{pub}})$ to $\mathcal{A}$.
      iv. 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: $\mathtt{pc} \leftarrow 0$.
      v. Increment the program counter: $\mathtt{pc} \leftarrow \mathtt{pc} + 1$ and go to Step 3.

Figure 2: Real Execution $\textsc{Real}_{\mathsf{RS},\mathcal{A},\mathcal{G}}(k)$

the next location. This is possible because our RAMs do not allow indirection as in loading e.g. $D[D[127]]$.

- **On the CRS.** In case no common reference string is required by the RAM scheme, we simply assume that $\mathbf{R}.\mathsf{Init}$ outputs the empty string. In such a case we sometimes avoid to write $\mathtt{crs}$ as input of $\mathbf{C}$, CPU and $\mathsf{Compute}$.

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

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

We write $\textsc{Ideal}_{\mathcal{S},\mathcal{G}}(k)$ for the output distribution in the ideal execution and we let $\textsc{Ideal}_{\mathcal{S},\mathcal{G}} = \{\textsc{Ideal}_{\mathcal{S},\mathcal{G}}(k)\}_{k\in\mathbb{N}}$.

Definition 4 (Security of a RAM Scheme)**.** *We say a RAM scheme* RS *is* $\mathbf{A}$*-secure (for function class* $\mathbb{G}$ *and program class* $\mathbb{P}$*) if* $\mathsf{RS}.\mathbf{C} : \mathbb{G} \to \mathbb{P}$ *and if for any function* $\mathcal{G} \in \mathbb{G}$ *and any* $\mathcal{A} \in \mathbf{A}$ *there exists a PPT simulator* $\mathcal{S}$ *such that* $\textsc{Real}_{\mathsf{RS},\mathcal{A},\mathcal{G}} \approx_c \textsc{Ideal}_{\mathcal{S},\mathcal{G}}$*.*

We introduce a notion of emulation, which facilitates designing compilers for less secure RAMs via compilers for more secure RAMs. We call a set $\mathbb{S}$ of RAM schemes a class if there exists $\mathbb{G}$ and $\mathbb{P}$ such that for all $\mathsf{RS} \in \mathbb{S}$ it holds that $\mathsf{RS}.\mathbf{C} : \mathbb{G} \to \mathbb{P}$. We write $\mathbb{S} : \mathbb{G} \to \mathbb{P}$. An emulator is a poly-time function $\mathcal{E} : \mathbb{S}_1 \to \mathbb{S}_2$, where $\mathbb{S}_1$ and $\mathbb{S}_2$ are RAM scheme classes $\mathbb{S}_1 : \mathbb{G} \to \mathbb{P}_1$ and

$\mathbb{S}_2 : \mathbb{G} \to \mathbb{P}_2$. I.e., given a RAM scheme $\mathsf{RS}_1 \in \mathbb{S}_1$ for some function class $\mathbb{G}$, the emulator outputs another RAM scheme $\mathsf{RS}_2 \in \mathbb{S}_2$ for the same function class.

Definition 5 (Secure Emulation)**.** *Let $\mathbb{S}_1 : \mathbb{G} \to \mathbb{P}_1$ and $\mathbb{S}_2 : \mathbb{G} \to \mathbb{P}_2$ be RAM scheme classes and let $\mathcal{E} : \mathbb{S}_1 \to \mathbb{S}_2$ be an emulator. We say that $\mathcal{E}$ is $(\mathbf{A}_1, \mathbf{A}_2)$-secure if for all $\mathsf{RS}_1 \in \mathbb{S}_1$ and $\mathsf{RS}_2 = \mathcal{E}(\mathsf{RS}_1)$ and $\mathcal{G} \in \mathbb{G}$ and all $\mathcal{A}_2 \in \mathbf{A}_2$ there exists a $\mathcal{A}_1 \in \mathbf{A}_1$ such that $\textsc{Real}_{\mathsf{RS}_1,\mathcal{A}_1,\mathcal{G}} \approx_c \textsc{Real}_{\mathsf{RS}_2,\mathcal{A}_2,\mathcal{G}}$.*

The following theorem is immediate.

Theorem 2. *Let $\mathcal{E} : \mathbb{S}_1 \to \mathbb{S}_2$ be an emulator. If $\mathcal{E}$ is $(\mathbf{A}_1, \mathbf{A}_2)$-secure and $\mathsf{RS}_1 \in \mathbb{S}_1$ is $\mathbf{A}_1$-secure, then $\mathsf{RS}_2 = \mathcal{E}(\mathsf{RS}_1)$ is $\mathbf{A}_2$-secure.*

4 Main Theorem

Our main result is a secure RAM scheme for the so-called split-state model, which we review below. This particular model can be cast as a special cases of our generic RAM model. We use $^{\mathsf{sp}}$ to denote the components of the split-state model, i.e., $\mathsf{RS}^{\mathsf{sp}} = (\mathbf{C}^{\mathsf{sp}}, \mathbf{R}^{\mathsf{sp}})$ and the adversary class is called $\mathbf{A}^{\mathsf{sp}}$.

In the split-state model we consider the secret disk D_{sec} split into two parts D_1 and D_2, and we require that leakage and tampering is done independently on the two parts. I.e., each position $D_{\mathsf{sec}}[i]$ on the secret disk is split into two parts $D_1[i]$ and $D_2[i]$ of equal length such that $D_{\mathsf{sec}}[i] = D_1[i] \| D_2[i]$. We let $D_1 = (D_1[2^{k-1}], \ldots, D_1[2^k - 1])$ and $D_2 = (D_2[2^{k-1}], \ldots, D_2[2^k - 1])$. The set $\mathbf{A}^{\mathsf{sp}}$ consists of all poly-time algorithms which never violate the following restrictions.

Tampering We require that a tampering function is of the form $\mathsf{Tamper}^{\mathsf{sp}} = (\mathsf{Tamper}_1^{\mathsf{sp}}, \mathsf{Tamper}_2^{\mathsf{sp}})$ and we let $\mathsf{Tamper}^{\mathsf{sp}}(D_{\mathsf{pub}} \| D_{\mathsf{sec}}) = D_{\mathsf{pub}} \| (\mathsf{Tamper}_1^{\mathsf{sp}}(D_1), \mathsf{Tamper}_2^{\mathsf{sp}}(D_2))$. Beside being split like this, there is no restriction on the tampering, i.e., each part of the secret disk can be arbitrarily tampered.

Disk leakage We also require that a disk leakage function is of the form $\mathsf{Leak}^{\mathsf{sp}} = (\mathsf{Leak}_1^{\mathsf{sp}}, \mathsf{Leak}_2^{\mathsf{sp}})$ and we let $\mathsf{Leak}^{\mathsf{sp}}(D_{\mathsf{pub}} \| D_{\mathsf{sec}}) = (\mathsf{Leak}_1^{\mathsf{sp}}(D_1), \mathsf{Leak}_2^{\mathsf{sp}}(D_2))$. Beside being split like this, we introduce a leakage bound $\mathsf{lb}_{\mathtt{disk}}$ and we require that the sum of the length of the leakage returned by all the leakage functions $\mathsf{Leak}_i^{\mathsf{sp}}$ is less than $\mathsf{lb}_{\mathtt{disk}}$.

Bus leakage We require that a bus leakage function is of the form $\mathsf{Leak}^{\mathsf{sp}} = (\mathsf{Leak}_1^{\mathsf{sp}}, \mathsf{Leak}_2^{\mathsf{sp}})$. For a bus $(i_0, D[i_0], i_1, D[i_1], \ldots, i_{1+2d}, D[i_{1+2d}])$ we let $B = (D[i_1], \ldots, D[i_{1+2d}])$ and we split B into two parts B_1 and B_2 by splitting each word, as done for the disk; the returned leakage is then $(i_0, i_1, i_2, \ldots, i_{1+2d}, \mathsf{Leak}_1^{\mathsf{sp}}(B_1), \mathsf{Leak}_2^{\mathsf{sp}}(B_2))$. Beside being split like this, we introduce a leakage bound $\mathsf{lb}_{\mathtt{bus}}$ and we require that the length of the leakage returned by each function $\mathsf{Leak}_i^{\mathsf{sp}}$ is less than $\mathsf{lb}_{\mathtt{bus}}$.

Note that by definition of the bus leakage, the CPU always leaks the program counter and the memory positions that are being read. Besides this it gives independent, bounded leakage on the parts of the words read up from the disk. Since the leakage and tamper classes for a split-state RAM are fully specified by $\mathsf{lb}_{\mathtt{disk}}$ and $\mathsf{lb}_{\mathtt{bus}}$ we will denote the adversary class for a split-state RAM simply by $\mathbf{A}^{\mathsf{sp}} = (\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathtt{bus}})$. Let $\mathbb{S}^{\mathsf{sp}}$ denote the class of split-state RAM schemes. We are now ready to state our main theorem.

Theorem 3 (Main Theorem)**.** *Let $\mathcal{C}$ be a $(\mathsf{lb}_{\mathtt{code}}, q)$-CNMLR code. There exists an efficient RAM scheme $\mathsf{RS} \in \mathbb{S}^{\mathsf{sp}}$ and a constant $c = O(1)$ such that RS is $(\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathtt{bus}})$-secure whenever $\mathsf{lb}_{\mathtt{disk}} + (c+1)\mathsf{lb}_{\mathtt{bus}} \leq \mathsf{lb}_{\mathtt{code}}$.*

The proof of the above theorem follows in two steps. We first define an intermediate model, which we call the hybrid model, where the adversary is only allowed a very limited form of leakage and tampering. For this model, we give a hybrid-to-split-state emulator (cf. Theorem 4 in Section 5). Then, we exhibit a RAM scheme that is secure in the hybrid model (cf. Theorem 5 in Section 6). Putting the above two things together with Theorem 2 concludes the proof of Theorem 3.

5 Hybrid-to-Split-State Emulator

We introduce an intermediate security model where the adversary is given only limited tampering/leakage capabilities. We call this model the *hybrid model*, and a RAM that is secure in this model is called a hybrid RAM; as for the split-state model, also the hybrid model can be cast as a special case of our generic RAM model. We use $^{\mathsf{hb}}$ to denote the components of the hybrid model, i.e., $\mathsf{RS}^{\mathsf{hb}} = (\mathbf{C}^{\mathsf{hb}}, \mathbf{R}^{\mathsf{hb}})$ and we call the adversary class $\mathbf{A}^{\mathsf{hb}}$.

5.1 The Hybrid Model

In the hybrid model the secret disk is not split. However, the tampering is very restricted: we only allow the adversary to copy values within the secret disk and to overwrite a location of the secret disk with a known value. In addition very little leakage is allowed. The adversary class $\mathbf{A}^{\mathsf{hb}}$ consists of all poly-time Turing machines never violating the following restrictions.

Tampering We require that each tampering function is a command of one of the following forms.

- If $\mathsf{Tamper} = (\texttt{COPY}, (j, j'))$ for $j, j' \geq 2^{k-1}$, then update $D[j'] \leftarrow D[j]$.
- If $\mathsf{Tamper} = (\texttt{REPLACE}, (j, \mathsf{val}))$ for $j \geq 2^{k-1}$ then update $D[j] \leftarrow \mathsf{val}$.

Disk leakage There is no other disk leakage from the secret disk, i.e., the adversary is not allowed any disk leakage queries.

Bus leakage There is only one allowed bus leakage function, say $\mathsf{Leak}^{\mathsf{hb}} = \mathsf{L}$, so this is by definition the leakage query used on each execution of the CPU. On this leakage query the adversary is given $(i_0, i_1, i_2, \ldots, i_{1+2d})$.

Note that by definition of the bus leakage, the CPU always leaks the program counter and the memory positions that are being read. Besides this it is given no leakage. Since the leakage and tamper classes for a hybrid RAM are implicitly specified, we will denote the adversary class for a hybrid RAM simply by $\mathbf{A}^{\mathsf{hb}}$.

Bounded-access schemes. We later want to compile programs for the hybrid model into more realistic models by encoding the positions in the disk using a code. Because of leakage from the bus, this only works if each value is not read up too many times. We therefore need a notion of a program for the hybrid model being c-bounding, meaning that such a program reads each value at most c times, even when the program is under attack by $\mathcal{A} \in \mathbf{A}^{\mathsf{hb}}$. To define this notion we use two vectors $\mathsf{Q}, \mathsf{C} \in \mathbb{N}^{2^k}$. If the value stored in $D[j]$ is necessarily known by the adversary, then $\mathsf{Q}[j] = \perp$. Otherwise, $\mathsf{Q}[j]$ will be an identifier for the possibly secret value stored in $D[j]$, and for an identifier $id = \mathsf{Q}[j]$ the value $\mathsf{C}[id]$ counts how many times the secret value with identifier id was accessed by the CPU. Initially $\mathsf{Q}[j] = \perp$ for all j and $\mathsf{C}[j] = 0$ for all j. After the initial encoding ω is stored, we set $\mathsf{Q}[2^{k-1} + j] = j$ for $j = 0, \ldots, |\omega_{\mathsf{sec}}| - 1$. Then let $\mathsf{ns} \leftarrow |\omega_{\mathsf{sec}}|$. We use this counter to remember the identifier for the next secret. During execution, when the adversary

executes $(\texttt{COPY}, (j, j'))$, then let $\mathsf{Q}[j'] = \mathsf{Q}[j]$. When the adversary executes $(\texttt{REPLACE}, (j, \mathsf{val}))$, then let $\mathsf{Q}[j] = \bot$. When the CPU executes, reading positions $i_0, i_1, \ldots, i_d$ and writing positions $j_1, \ldots, j_d$ then proceed as follows. For $p = 0, \ldots, d$, if $\mathsf{Q}[i_p] \neq \bot$, let $\mathsf{C}[\mathsf{Q}[i_p]] \leftarrow \mathsf{C}[\mathsf{Q}[i_p]] + 1$. Then proceed as follows. If $\mathsf{Q}[i_0] = \mathsf{Q}[i_1] = \cdots = \mathsf{Q}[i_d] = \bot$, then let $\mathsf{Q}[j_1] = \cdots = \mathsf{Q}[j_d] = \bot$. Otherwise, let $(\mathsf{Q}[j_1], \ldots, \mathsf{Q}[j_d]) = (\mathsf{ns}, \ldots, \mathsf{ns} + d - 1)$ and let $\mathsf{ns} \leftarrow \mathsf{ns} + d$. Then for each $j_i < 2^{k-1}$, set $\mathsf{Q}[j_i] \leftarrow \bot$.

We say that a hybrid RAM scheme RS is c-bounding if it holds for all $\mathcal{G} \in \mathsf{RS}.\mathbf{C}.\mathbb{G}$ that if $\mathsf{RS}.\mathbf{C}(\mathcal{G})$ is executed on $\mathsf{RS}.\mathbf{R}$ under attack by $\mathcal{A} \in \mathbf{A}^{\mathsf{hb}}$ and the above vectors are computed during the attack, then it never happens that $\mathsf{C}[j] > c$ for any j. Let $\mathbb{G}$ denote the class of poly-time functionalities. We use $\mathbb{S}_c^{\mathsf{hb}} : \mathbb{G} \to \mathbb{P}_c^{\mathsf{hb}}$ to denote the class of hybrid RAM schemes which are c-bounding.

Theorem 4. *Let $\mathcal{C}$ be a $(\mathsf{lb}_{\mathtt{code}}, q)$-CNMLR code. Let $\mathbf{A}^{\mathsf{sp}} = (\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathtt{bus}})$ be a split-state adversary class such that $\mathsf{lb}_{\mathtt{disk}} + (c+1) \cdot \mathsf{lb}_{\mathtt{bus}} \leq \mathsf{lb}_{\mathtt{code}}$. Then there exists an $(\mathbf{A}^{\mathsf{hb}}, \mathbf{A}^{\mathsf{sp}})$-secure emulator $\mathcal{E} : \mathbb{S}_c^{\mathsf{hb}} \to \mathbb{S}^{\mathsf{sp}}$.*

5.2 The Emulator

The proof of Theorem 4 can be found in Appendix A; here we provide only a high-level overview. The goal of the emulator $\mathcal{E}$ is to transform a hybrid RAM scheme $\mathsf{RS}^{\mathsf{hb}} = (\mathbf{C}^{\mathsf{hb}}, \mathbf{R}^{\mathsf{hb}}) \in \mathbb{S}_c^{\mathsf{hb}}$ into a split-state RAM scheme $\mathcal{E}(\mathsf{RS}^{\mathsf{hb}}) = \mathsf{RS}^{\mathsf{sp}} = (\mathbf{C}^{\mathsf{sp}}, \mathbf{R}^{\mathsf{sp}})$. In particular, the emulator needs to specify transformations for the components of $\mathsf{RS}^{\mathsf{hb}}$. This includes the contents of the disk as well as the way instructions are stored and processed by the CPU. Below, we give an overview of the construction of the emulator.

We emulate a program as follows $\mathcal{E}(\ell_P, I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}, \omega^{\mathsf{hb}}) = (\ell_P, I, \ell_I, O, \ell_O, \mathcal{X}, \mathcal{Y}, \omega^{\mathsf{sp}})$, where we simply let $\omega_{\mathsf{pub}}^{\mathsf{sp}}$ be $\omega_{\mathsf{pub}}^{\mathsf{hb}}$. Then for each $j \in [0, |\omega_{\mathsf{sec}}^{\mathsf{hb}}|]$, let $\omega_{\mathsf{sec}}^{\mathsf{sp}}[j] = (\omega_{\mathsf{sec},1}^{\mathsf{sp}}[j], \omega_{\mathsf{sec},2}^{\mathsf{sp}}[j])$ be an encoding of $\omega_{\mathsf{sec}}^{\mathsf{hb}}[j]$ (computed using a CNMLR code, see Section 2). The CPU $\mathsf{Compute}^{\mathsf{sp}}$ runs as follows. It reads up the same instruction $D^{\mathsf{hb}}[\mathsf{pc}]$ that $\mathsf{Compute}^{\mathsf{hb}}$ would. Then for each additional position $D^{\mathsf{hb}}[i]$ read up, if $i < 2^{k-1}$ it lets $v_i = D^{\mathsf{hb}}[i]$ and if $i \geq 2^{k-1}$ it lets $(v_{1,i}, v_{2,i}) = D^{\mathsf{hb}}[i]$ and decodes $(v_{1,i}, v_{2,i})$ to v_i. If any decoding fails, then $\mathsf{Compute}^{\mathsf{sp}}$ self-destructs. Otherwise it runs $\mathsf{Compute}^{\mathsf{hb}}$ on the v_j values. Finally, it encodes all values v_j to be stored on $D_{\mathsf{sec}}^{\mathsf{sp}}$ and writes them back to disk. Then values v_j to be stored on $D_{\mathsf{pub}}^{\mathsf{sp}}$ are stored in "plaintext" as v_j.

Security of emulation. To argue security of emulation, we need to show that for all adversaries $\mathcal{A} \in \mathbf{A}^{\mathsf{sp}}$ there exists a simulator $\mathcal{B} \in \mathbf{A}^{\mathsf{hb}}$ able to fake $\mathcal{A}$'s view in a real execution with $\mathsf{RS}^{\mathsf{sp}}$ given only its limited leakage/tampering capabilities (via $\texttt{REPLACE}$ and $\texttt{COPY}$ commands). The simulator $\mathcal{B}$ runs $\mathcal{A}$ as a sub-routine, and works in two phases: the pre-processing and the online phase. Initially, in the pre-processing $\mathcal{B}$ samples $\texttt{crs}$ and creates encodings of 0 for all the values on the secret disk using the CNMLR code, and puts dummy encodings $(v_1, v_2) \leftarrow \mathsf{Encode}(\texttt{crs}, 0)$ on the corresponding simulated virtual disks. For the positions on the public disk, the simulator can put the correct values, which is possible as it can read $\omega_{\mathsf{pub}}^{\mathsf{hb}}$ from $D_{\mathsf{pub}}^{\mathsf{hb}}$ and $\omega_{\mathsf{pub}}^{\mathsf{hb}} = \omega_{\mathsf{pub}}^{\mathsf{sp}}$. Depending on the queries in the online phase $\mathcal{B}$ will update these virtual disks in the following. $\texttt{TAMPER}$ queries are simulated easily by applying the corresponding tamper functions to the current state of the virtual disks D_1 and D_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 $\texttt{EXEC}$ 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 disk that are used by the CPU to evaluate the current instruction are *public* (corresponding to the case $\mathsf{Q}[j_1] = \cdots = \mathsf{Q}[j_d] = \perp$ in the definition of c-bounded) and the case where 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 the CPU (including the outputs and the new contents of the disks). If, on the other hand, some values that are used by the CPU in the current execution are secret, then $\mathcal{B}$'s only chance to simulate $\mathcal{A}$ is to run $\mathsf{CPU}^{\mathsf{hb}}$ in the hybrid game. The difficulty is to keep the state of the secret hybrid disk D^{hb} consistent with the contents of the virtual disks D_1, D_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 D^{hb}. The simulator $\mathcal{B}$ manages this book-keeping by using two records: (i) the vector S that stores dummy encodings (v_1, v_2) corresponding to values unknown to $\mathcal{B}$ (either generated during the pre-processing, or resulting from an evaluation of $\mathsf{CPU}^{\mathsf{hb}}$ on partially secret inputs); (ii) the backup storage $\mathcal{BP}$ that $\mathcal{B}$ maintains on the hybrid disk D^{hb} that stores a copy of all values that are unknown to the adversary (essentially, the values on $\mathcal{BP}$ correspond to the values that the dummy encodings in S where supposed to encode). Then the simulator can always copy the corresponding secret value to the position on D^{hb}, which corresponds to the value that *should* have been inside the encoding on the same position on the two virtual disks. The trick is that each secret value, i.e., a value that would have an identifier in the definition of c-boundedness, has an associated dummy encoding generated by the simulator and a corresponding value on $D^{\mathsf{hb}}_{\mathsf{pub}}$. The simulator uses the book-keeping to keep these values "lined up". All other encodings were not generated by the simulator, and can therefore be decoded to values independent of the values in the dummy encodings. These therefore correspond to public values. A reduction to continuous non-malleability then allows to replace the 0's in the dummy encoding by the correct values on D^{hb}.

6 The Hybrid Scheme

In this section we describe an $O(1)$-bounding, RAM scheme $\mathsf{RS}^{\mathsf{hb}} = (\mathbf{C}^{\mathsf{hb}}, \mathbf{R}^{\mathsf{hb}})$ that is secure in the hybrid model. Recall that a hybrid schemes $\mathsf{RS}^{\mathsf{hb}}$ consists of a hybrid RAM $\mathbf{R}^{\mathsf{hb}}$ and a hybrid compiler $\mathbf{C}^{\mathsf{hb}}$ which takes a functionality $\mathcal{G}$ with secret key K and outputs an encoding of the form $(P, \omega^{\mathsf{hb}})$ to be executed on $\mathbf{R}^{\mathsf{hb}}$. The RAM $\mathbf{R}^{\mathsf{hb}}$ consists of a CPU $\mathsf{CPU}^{\mathsf{hb}}$, which is specified by two functions $\mathsf{Random}^{\mathsf{hb}}$ and $\mathsf{Compute}^{\mathsf{hb}}$. Below, we present an outline of our hybrid RAM scheme $\mathsf{RS}^{\mathsf{hb}}$ and refer the reader to Appendix B for the details.

Overview. We assume $\mathcal{G}$ is described by 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 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 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 is 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 Appendix B.1 for the complete specifications.

Let $\mathbb{G}$ be the class of poly-time keyed functions $\mathcal{G}$. (each described a regular program as outlined above). We show the following theorem.

Theorem 5. *There exists an* $\mathbf{A}^{\mathsf{hb}}$*-secure RAM scheme* $\mathsf{RS}^{\mathsf{hb}} = (\mathbf{C}^{\mathsf{hb}}, \mathbf{R}^{\mathsf{hb}})$ *for function class* $\mathbb{G}$ *and program class* $\mathbb{P}_c^{\mathsf{hb}}$ *for* $c = O(1)$.

The hybrid scheme. Our hybrid compiler $\mathbf{C}^{\mathsf{hb}}$ takes as input $\mathcal{G} \in \mathbb{G}$ and is supposed to produce a *compiled* program (during the pre-processing phase) to be run by the hybrid RAM $\mathbf{R}^{\mathsf{hb}}$ (during the on-line phase). The compiled program is placed on the disk from which $\mathsf{CPU}^{\mathsf{hb}}$ reads in sequence. Our CPU $\mathsf{CPU}^{\mathsf{hb}} = (\mathsf{Compute}^{\mathsf{hb}}, \mathsf{Random}^{\mathsf{hb}})$ will be deterministic, and hence $\mathsf{Random}^{\mathsf{hb}}$ just outputs the empty string at each invocation. This means that we only have to specify the compiler $\mathbf{C}^{\mathsf{hb}}$ and the function $\mathsf{Compute}^{\mathsf{hb}}$ for a complete specification of $\mathsf{RS}^{\mathsf{hb}}$.

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 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 $(\mathtt{ac}, \mathtt{pc})$ 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., trying to reset the CPU by forcing it the CPU to re-use a previously encoded value).

Whenever algorithm $\mathsf{Compute}^{\mathsf{hb}}$ 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 Appendix B.2.

Analysis. Next, we turn to a high-level overview of the security proof (the actual proof can be found in Appendix B.3). Our goal is to prove that the above RAM scheme is secure in the hybrid model, namely for all adversaries $\mathcal{B} \in \mathbf{A}^{\mathsf{hb}}$ attacking the RAM scheme in a real 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 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 disk using a "replace" command do not contain the secret label. In each execution our CPU $\mathsf{CPU}^{\mathsf{hb}}$ 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 containing the secret label in the augmentation were written by the pre-processing or by $\mathsf{CPU}^{\mathsf{hb}}$, 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 $\mathsf{CPU}^{\mathsf{hb}}$ on values known by the adversary. We then argue that $\mathsf{CPU}^{\mathsf{hb}}$ (by design) will never write two values $V \neq V'$ sharing the same augmentation

(j, L, a, p). This is because the augmentation includes the strictly increasing pair (a, p), and we also prove that $\mathsf{CPU}^{\mathsf{hb}}$ can predict what (a, p) should be for all loaded values in all executions. It follows from an inductive argument that all values containing the secret label in the augmentation are correct. Hence all values on the disk are either correct secret values or incorrect values known by the adversary. So, when $\mathsf{CPU}^{\mathsf{hb}}$ writes a result to the disk, it is either an allowed output or a value already known by the adversary. From the above intuition, it is straight-forward, although rather tedious, to derive a simulator.

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 and perturbations. *IACR Cryptology ePrint Archive*, 2014:316, 2014.

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

[4] Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In *EUROCRYPT*, pages 491–506, 2003.

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

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

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

[8] Sandro Coretti, Yevgeniy Dodis, Björn Tackmann, and Daniele Venturi. Self-destruct non-malleability. Cryptology ePrint Archive, Report 2014/866, 2014. `http://eprint.iacr.org/`.

[9] Sandro Coretti, Ueli Maurer, Björn Tackmann, and Daniele Venturi. From single-bit to multi-bit public-key encryption via non-malleable codes. In *TCC*, 2015. To appear.

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

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

[12] Dana Dachman-Soled, Feng-Hao Liu, Elaine Shi, and Hong-Sheng Zhou. Locally decodable and updatable non-malleable codes and their applications. In *TCC*, 2015. To appear.

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

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

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

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

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

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

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

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

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

Pre-processor: The pre-processor $\mathbf{C}^{\mathsf{sp}} = \mathcal{E}(\mathbf{C}^{\mathsf{hb}})$ runs as follows: Sample $(P, \omega^{\mathsf{hb}}_{\mathsf{pub}}, \omega^{\mathsf{hb}}_{\mathsf{sec}}) \leftarrow \mathbf{C}^{\mathsf{hb}}(\mathtt{crs}, (\mathcal{G}, \mathsf{K}))$. Output $(P, \omega^{\mathsf{sp}}_{\mathsf{pub}}, \omega^{\mathsf{sp}}_{\mathsf{sec}})$, where the content of the public section $\omega^{\mathsf{sp}}_{\mathsf{pub}}$ and the secret section $\omega^{\mathsf{sp}}_{\mathsf{sec}} = (\omega^{\mathsf{sp}}_{\mathsf{sec},1}, \omega^{\mathsf{sp}}_{\mathsf{sec},2})$ are detailed below.

Secret Disk: The secret disk of the hybrid pre-processing is encoded by encoding each memory position using the non-malleable code: For $i = 0, \ldots, |\omega^{\mathsf{hb}}_{\mathsf{sec}}| - 1$, let $v_i = \omega^{\mathsf{hb}}_{\mathsf{sec}}[i]$, sample $(v_{1,i}, v_{2,i}) \leftarrow \mathsf{Encode}(\mathtt{crs}, v_i)$ and set $\omega^{\mathsf{sp}}_{\mathsf{sec},1}[i] = v_{1,i}$ and set $\omega^{\mathsf{sp}}_{\mathsf{sec},2}[i] = v_{2,i}$.

Public Disk: Let $\omega^{\mathsf{sp}}_{\mathsf{pub}} = \omega^{\mathsf{hb}}_{\mathsf{pub}}$. For an instruction of the compiled form $(\mathsf{Y}, \mathsf{I}, \mathsf{O}) = \mathcal{E}(\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}})$, we let $\mathcal{E}^{-1}(\mathsf{Y}, \mathsf{I}, \mathsf{O}) = (\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}})$. 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 $(\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}})$. Note that if and when the CPU of the hybrid scheme reads up $\mathtt{sd}$ it will self-destruct, see Fig. 1.

CPU: The compiled CPU $(\mathsf{Random}^{\mathsf{sp}}, \mathsf{Compute}^{\mathsf{sp}}) = \mathcal{E}(\mathsf{Random}^{\mathsf{hb}}, \mathsf{Compute}^{\mathsf{hb}})$ works as follows: $\mathsf{Random}^{\mathsf{sp}} = \mathsf{Random}^{\mathsf{hb}}$ and $((\mathtt{O}_1, \ldots, \mathtt{O}_d), \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{Compute}^{\mathsf{sp}}(\mathtt{crs}, (\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_d), r, \mathtt{pc}, \mathtt{ac})$ is specified by:

1. If $\mathcal{E}^{-1}(\mathtt{R}_0) = \mathtt{sd}$, then self-destruct. Otherwise, compute $(\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}}) = \mathcal{E}^{-1}(\mathtt{R}_0)$, and set $\mathtt{R}^{\mathsf{hb}}_0 = (\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}})$.
2. For $j = 1 \ldots, d$, let $\mathtt{loc}^{\mathsf{hb}}_j = \mathsf{I}^{\mathsf{hb}}[j]$. If $\mathtt{loc}^{\mathsf{hb}}_j < 2^{k-1}$, then let $\mathtt{R}^{\mathsf{hb}}_j = \mathtt{R}_j$. If $\mathtt{loc}^{\mathsf{hb}}_j \geq 2^{k-1}$, then let $(v_{1,j}, v_{2,j}) = \mathtt{R}_j$. Let $v_j = \mathsf{Decode}(\mathtt{crs}, (v_{1,j}, v_{2,j}))$. If $v_j = \perp$, then self-destruct. Otherwise, let $\mathtt{R}^{\mathsf{hb}}_j = v_j$.
3. Compute $((\mathtt{O}^{\mathsf{hb}}_1, \ldots, \mathtt{O}^{\mathsf{hb}}_d), \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{Compute}((\mathtt{R}^{\mathsf{hb}}_0, \mathtt{R}^{\mathsf{hb}}_1, \ldots, \mathtt{R}^{\mathsf{hb}}_d), r, \mathtt{pc}, \mathtt{ac})$.
4. For $j = 1 \ldots, d$, let $\mathtt{loc}_j = \mathsf{O}^{\mathsf{hb}}[j]$. If $\mathtt{loc}^{\mathsf{hb}}_j < 2^{k-1}$, then set $\mathtt{O}_j \leftarrow \mathtt{O}^{\mathsf{hb}}_j$. If $\mathtt{loc}^{\mathsf{hb}}_j \geq 2^{k-1}$, then let $v_j = \mathtt{O}^{\mathsf{hb}}_j$, sample $(v_{1,j}, v_{2,j}) \leftarrow \mathsf{Encode}(\mathtt{crs}, v_j)$, and let $\mathtt{O}^{\mathsf{hb}}_j \leftarrow (v_{1,j}, v_{2,j})$.

Dimensions: The length of the public sectors of the disks will be the same. The size of the secret disk of the split-state RAM is double the size of the one for the hybrid RAM. The word size of the produced RAM will be large enough to hold an encoding under $\mathcal{C}$ as produced by the underlying non-malleable code.

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

A 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 conclude the proof we need to show that the view produced by the hybrid simulator $\mathcal{B} \in \mathbf{A}^{\mathsf{hb}}$, interacting with $\mathcal{A}$ in $\text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B},\mathcal{G}}(k)$, is computationally indistinguishable from the view that $\mathcal{A} \in \mathbf{A}^{\mathsf{sp}}$ obtains in a real execution $\text{REAL}_{\mathsf{RS}^{\mathsf{sp}},\mathcal{A},\mathcal{G}}(k)$. 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 $\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{R}}(b)$ (where b is a random bit). 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 $(P,\omega^{\mathsf{hb}}_{\mathsf{pub}},\omega^{\mathsf{hb}}_{\mathsf{sec}}) \leftarrow \mathbf{C}^{\mathsf{hb}}(\mathcal{G},\mathsf{K})$. 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 $\omega^{\mathsf{hb}}_{\mathsf{sec}}$ (and hence with the simulated virtual disks D_1, D_2). $\mathcal{R}$ uses so-called *disk reconstruction functions* 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 D_i. Given such functions 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 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 $(\mathsf{Tamper}_1, \mathsf{Tamper}_2)$ then Dcon_i is updated by concatenating Tamper_i with the current Dcon_i, i.e., we get $\mathsf{Dcon}'_i = \mathsf{Dcon}_i \circ \mathsf{Tamper}_i$ for all $i \in \{1,2\}$. The full details about how $\mathcal{R}$ maintains Dcon_i are given in Fig. 6-7.

One can verify that if in the simulation we initialize the secret disks D_1, D_2 with encodings of the correct secret values, then the simulator $\mathcal{B}$ from Fig. 4–5 produces exactly the distribution as in $\text{REAL}_{\mathsf{RS}^{\mathsf{sp}},\mathcal{A},\mathcal{G}}$. Hence, the reduction will essentially run the code of $\mathcal{B}$ and submit to its target oracles inputs of the form $(0,\omega^{\mathsf{hb}}[j])$ in each iteration of the loop. Depending on the challenge bit b, the reduction either simulates $\text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B},\mathcal{G}}$ (if $b=0$) or $\text{REAL}_{\mathsf{RS}^{\mathsf{sp}},\mathcal{A},\mathcal{G}}$ (if $b=1$), i.e., we have:

$$\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{R}}(0) \equiv \text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B},\mathcal{G}}$$
$$\text{GAME}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{R}}(1) \equiv \text{REAL}_{\mathsf{RS}^{\mathsf{sp}},\mathcal{A},\mathcal{G}}.$$

Security of the emulator $\mathcal{E}$ now follows from the adaptive composability of the underlying CNMLR code (see Theorem 1).

Computing the leakage bound. We finally argue about why our reduction satisfies the leakage bounds of the target oracles. First, observe that since $\mathsf{RS}^{\mathsf{hb}}$ 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 $\mathsf{lb}_{\mathsf{bus}}$ bits, for a total of $c \cdot \mathsf{lb}_{\mathsf{bus}}$. Furthermore, each encoding might enter into the leakage queries to

Pre-processing: The hybrid game will sample $(P, \omega^{\mathsf{hb}}_{\mathsf{pub}}, \omega^{\mathsf{hb}}_{\mathsf{sec}}) \leftarrow \mathbf{C}^{\mathsf{hb}}(\mathcal{G}, \mathsf{K})$, give $(P, \omega^{\mathsf{hb}}_{\mathsf{pub}})$ to the simulator $\mathcal{B}$, and store ω^{hb} on D^{hb}. The simulator passes $(P, \omega^{\mathsf{hb}}_{\mathsf{pub}})$ to $\mathcal{A}$. The simulator $\mathcal{B}$ has to create simulated disks (D_1, D_2) defining $D^{\mathsf{sp}}_{\mathsf{sec}}$. This is done as follows. Sample $\mathtt{crs} \leftarrow \mathsf{Init}(1^k)$. For $j = 0, \ldots, |\omega^{\mathsf{hb}}_{\mathsf{sec}}| - 1$, sample $(v_{1,j}, v_{2,j}) \leftarrow \mathsf{Encode}(\mathtt{crs}, 0)$, let $\mathsf{S}[j] = (v_{1,j}, v_{2,j})$, $D_1[j] = v_{1,j}$ and $D_2[j] = v_{2,j}$, and then choose a backup location $\mathcal{BP}(j)$ on the secret disk such that $D^{\mathsf{hb}}_{\mathsf{sec}}[\mathcal{BP}(j)]$ is never accessed by $\mathsf{RS}^{\mathsf{hb}}$ or $\mathcal{A}$, and issue the command $(\mathtt{COPY}, j, \mathcal{BP}(j))$ to create a back up of the value $D^{\mathsf{hb}}_{\mathsf{sec}}[j] = \omega^{\mathsf{hb}}_{\mathsf{sec}}[j]$. Notice that in this way the simulator $\mathcal{B}$ keeps a copy of the original secret value that $(v_{1,j}, v_{2,j})$ is supposed to encode (instead of 0 as after the pre-processing of $\mathcal{B}$). Finally, let $\mathsf{ns} = |\omega^{\mathsf{hb}}_{\mathsf{sec}}|$.

Figure 4: The Simulator, $\mathcal{B}^{\mathcal{A}}$, pre-processing

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 $\mathsf{RS}^{\mathsf{hb}}$ 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 Leak'_i needs to compute B_i. The value B_i contains values written on the bus during the reading phase and the writing phase.

Let us start by discussing the writing phase, as this is the easier case. Here Leak'_i computes the disk $D'_i = \mathsf{Dcon}'_i(\mathbf{c}_i)$ as it looked after the writing phase, and adds to B_i each $v_i = D_i[\mathtt{loc}^{\mathsf{hb}}_j]$ from the writing phase. Notice that, however, for each $D_i[\mathtt{loc}^{\mathsf{hb}}_j]$ there exists an index g defined in Step 4d such that $D_i[\mathtt{loc}^{\mathsf{hb}}_j] = \mathbf{c}_i[g]$ 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 B_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 model the counter $\mathsf{C}[g]$ is set to 1. In particular, the leakage tally of $\mathbf{c}_i[g]$ is less than $\mathsf{C}[g] \cdot \mathsf{lb}_{\mathtt{bus}}$.

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 B_i each value $v_i = D_i[\mathtt{loc}^{\mathsf{hb}}_j]$. 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 $\mathsf{T}_i(\mathbf{c}_i) = \mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}^{\mathsf{hb}}_j]$. The reply is either $(\mathsf{same}^\star, g)$, $\perp$ or a 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 $(v_1, v_2) = (\mathsf{Dcon}_1(\mathbf{c}_1)[\mathtt{loc}^{\mathsf{hb}}_j], \mathsf{Dcon}_2(\mathbf{c}_2)[\mathtt{loc}^{\mathsf{hb}}_j])$. Hence the reduction can hard-code the value v_i into the leakage query Leak_i and add it to B_i as the value $D_i[\mathtt{loc}^{\mathsf{hb}}_j] = v_i$.
- If the reply was $(\mathsf{same}^\star, g)$, then by construction of the function $\mathsf{T}_i(\mathbf{c}_i) = \mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}^{\mathsf{hb}}_j]$, we have that $(D_1[\mathtt{loc}^{\mathsf{hb}}_j], D_2[\mathtt{loc}^{\mathsf{hb}}_j])$ 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.[12] So, the leakage function

[12]Here we use the 'stronger' property of the CNMC code which ensures that the oracle returns not only the symbol

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 $\texttt{CMD} = (\texttt{LEAK}, (\mathsf{Leak}_1, \mathsf{Leak}_2))$, proceed as follows. Compute $\Lambda \leftarrow \mathsf{Leak}(D^{\mathsf{sp}}_{\mathsf{pub}}, D_1, D_2)$ using the simulated virtual disks and give Λ to $\mathcal{A}$.
3. If $\texttt{CMD} = (\texttt{TAMPER}, (\mathsf{Tamper}_1, \mathsf{Tamper}_2))$ then proceed as follows. Compute $(D^{\mathsf{sp}}_{\mathsf{pub}}, D_1, D_2) \leftarrow (D^{\mathsf{sp}}_{\mathsf{pub}}, \mathsf{Tamper}_1(D_1), \mathsf{Tamper}_2(D_2))$.
4. If $\texttt{CMD} = (\texttt{EXEC}, (\mathsf{Leak}_1, \mathsf{Leak}_2), D')$ and $\texttt{B} = 0$ then do the following:
   (a) Let $D^{\mathsf{sp}}_{\mathsf{pub}} \leftarrow D'$. If $\mathcal{E}^{-1}(D^{\mathsf{sp}}[\texttt{pc}]) = \texttt{sd}$, then self-destruct. Otherwise, compute $(\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}}) = \mathcal{E}^{-1}(D^{\mathsf{sp}}[\texttt{pc}])$, and set $\texttt{R}^{\mathsf{hb}}_0 = (\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}})$.
   (b) For $j = 1 \ldots, d$, let $\texttt{loc}^{\mathsf{hb}}_j = \mathsf{I}^{\mathsf{hb}}[j]$. If $\texttt{loc}^{\mathsf{hb}}_j < 2^{k-1}$, then let $\texttt{R}^{\mathsf{hb}}_j = D^{\mathsf{sp}}[\texttt{loc}_j]$. If $\texttt{loc}^{\mathsf{hb}}_j \geq 2^{k-1}$, then let $(v_{1,j}, v_{2,j}) = (D_1[\texttt{loc}^{\mathsf{hb}}_j], D_2[\texttt{loc}^{\mathsf{hb}}_j])$. If $\exists g : (v_{1,j}, v_{2,j}) = \mathsf{S}[g]$, then issue the command $(\texttt{COPY}, (\mathcal{BP}(g), \texttt{loc}^{\mathsf{hb}}_j))$ to put back in $D^{\mathsf{hb}}[\texttt{loc}^{\mathsf{hb}}_j]$ the value that $(v_{1,j}, v_{2,j})$ *should* have been an encoding of. If $\nexists g : (v_{1,j}, v_{2,j}) = \mathsf{S}[g]$, then compute $v_j = \mathsf{Decode}(\texttt{crs}, v_{1,j}, v_{2,j})$. If $v_j = \perp$, then simulate a self-destruct (by ignoring all future $\texttt{EXEC}$ commands). Otherwise, issue the command $(\texttt{REPLACE}, (\texttt{loc}^{\mathsf{hb}}_j, v_j))$ to put in $D^{\mathsf{hb}}[\texttt{loc}^{\mathsf{hb}}_j]$ the value that $(v_{1,j}, v_{2,j})$ is an encoding of.
   (c) Issue the command $\texttt{CMD} = (\texttt{EXEC}, D')$ to run $\mathsf{CPU}^{\mathsf{hb}}$, which replaces the existing disks by the modified output: $(D^{\mathsf{hb}}_{\mathsf{pub}}, D^{\mathsf{hb}}_{\mathsf{sec}}, \texttt{B}, \texttt{T}) \leftarrow \mathsf{CPU}^{\mathsf{hb}}(D^{\mathsf{hb}}_{\mathsf{pub}}, D^{\mathsf{hb}}_{\mathsf{sec}}, \texttt{pc}, \texttt{ac})$. This also updates $D^{\mathsf{sp}}_{\mathsf{pub}}$ as we identify $D^{\mathsf{sp}}_{\mathsf{pub}} = \mathcal{E}(D^{\mathsf{hb}}_{\mathsf{pub}})$.
   (d) Update the simulated disks D_1 and D_2. For $j = 1 \ldots, d$, let $\texttt{loc}^{\mathsf{hb}}_j = \mathsf{O}^{\mathsf{hb}}[j]$. How we process each $\texttt{loc}^{\mathsf{hb}}_j$ depends on whether the above execution was a *secret execution* (i.e., $\mathsf{Q}[\texttt{loc}^{\mathsf{hb}}_1] = \ldots = \mathsf{Q}[\texttt{loc}^{\mathsf{hb}}_d] = \perp$ see Section 5.1) or not.

   public: If $\texttt{loc}^{\mathsf{hb}}_j \geq 2^{k-1}$, then let $v_j = D^{\mathsf{hb}}[\texttt{loc}^{\mathsf{hb}}_j]$,[a] sample $(v_{1,j}, v_{2,j}) \leftarrow \mathsf{Encode}(\texttt{crs}, v_j)$ and let $(D_1[\texttt{loc}^{\mathsf{hb}}_j], D_2[\texttt{loc}^{\mathsf{hb}}_j]) \leftarrow (v_{1,j}, v_{2,j})$.

   secret: If $\texttt{loc}^{\mathsf{hb}}_j \geq 2^{k-1}$, then sample $(v_{1,j}, v_{2,j}) \leftarrow \mathsf{Encode}(\texttt{crs}, 0)$ and let $(D_1[\texttt{loc}^{\mathsf{hb}}_j], D_2[\texttt{loc}^{\mathsf{hb}}_j]) \leftarrow (v_{1,j}, v_{2,j})$. Then let $\mathsf{S}[\mathsf{ns}] \leftarrow (v_{1,j} v_{2,j})$, pick a fresh back-up location $\mathcal{BP}(\mathsf{ns})$, issue the command $(\texttt{COPY}, \texttt{loc}^{\mathsf{hb}}_j, \mathcal{BP}(\mathsf{ns}))$ to back up the value that $(v_{1,j}, v_{2,j})$ *should* have been an encoding of, and then let $\mathsf{ns} \leftarrow \mathsf{ns} + 1$.

   (e) Finally simulate the leakage $\Lambda_1 = \mathsf{Leak}_1(B_1)$ and $\Lambda_2 = \mathsf{Leak}_2(B_2)$ by computing $\mathsf{Bs} = (B_1, B_2)$ as in a real execution, but using the above simulated values. In particular, $B_i \ni v_j$ for all $(v_{1,j}, v_{2,j}) = (D_1[\texttt{loc}^{\mathsf{hb}}_j], D_2[\texttt{loc}^{\mathsf{hb}}_j])$ from the reading and all $(v_{1,j}, v_{2,j}) \leftarrow \mathsf{Encode}(\texttt{crs}, 0)$ 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 thus $D^{\mathsf{hb}}[\texttt{loc}_j]$. 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 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

Pre-processing: Sample $(P, \omega^{\mathsf{hb}}_{\mathsf{pub}}, \omega^{\mathsf{hb}}_{\mathsf{sec}}) \leftarrow \mathbf{C}^{\mathsf{hb}}(\mathcal{G}, \mathsf{K})$ and give $(P, \omega^{\mathsf{hb}}_{\mathsf{pub}})$ to $\mathcal{A}$. Let $D^{\mathsf{sp}}_{\mathsf{pub}} = \mathcal{E}(\omega^{\mathsf{hb}}_{\mathsf{pub}})$ and $D^{\mathsf{hb}}_{\mathsf{sec}} = \omega^{\mathsf{hb}}_{\mathsf{sec}}$. Create virtual disks $D_1 = \omega^{\mathsf{sp}}_{\mathsf{sec},1}$ and $D_2 = \omega^{\mathsf{sp}}_{\mathsf{sec},2}$ "inside the leakage oracles" $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_1, \cdot)$ and $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_2, \cdot)$ by maintaining *disk reconstruction functions* Dcon_1, Dcon_2 such that $D_i = \mathsf{Dcon}_i(\mathbf{c}_i)$. Initially Dcon_i is the function outputting 0 on all inputs j. We elaborate on how to maintain Dcon_i below. For $j = 0, \ldots, |\omega^{\mathsf{hb}}_{\mathsf{sec}}| - 1$, output $(0, \omega^{\mathsf{hb}}_{\mathsf{sec}}[j])$ to the game $\textsc{Game}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{R}}(b)$ to make it create an encoding $(v_{1,j}, v_{2,j})$ of either 0 or $\omega^{\mathsf{hb}}_{\mathsf{sec}}[j]$ and add $v_{1,j}$ to $\mathbf{c}_1[j]$ and $v_{2,j}$ to $\mathbf{c}_2[j]$.

Notice that the record S kept by $\mathcal{B}$ in Figure 4 is now represented by $(\mathbf{c}_1, \mathbf{c}_2)$: We will maintain the invariant that whenever $\mathcal{B}$ samples (v_1, v_2) and stores it in $\mathsf{S}[g]$, the reduction makes an encoding query to its challenge oracle as specified in Definition 2, and this will be the g-th such query, such that $(\mathbf{c}_1[g], \mathbf{c}_2[g]) = (v_1, v_2)$.

Update the disk reconstruction functions Dcon_1, Dcon_2 as follows: Let Dcon_i be the function before the above initial encodings were computed; set $\mathsf{Dcon}'_i = \mathsf{Dcon}_i$, except that $(\mathsf{Dcon}'_i(\mathbf{c}_i))[j] = \mathbf{c}_i[j]$ for $j = 0, \ldots, |\omega^{\mathsf{hb}}_{\mathsf{sec}}| - 1$. Furthermore, for each j, let $\mathcal{BP}(j)$ be the back-up location chosen by $\mathcal{B}$; simulate the commands $(\texttt{COPY}, (j, \mathcal{BP}(j)))$ by setting $D^{\mathsf{hb}}_{\mathsf{sec}}[\mathcal{BP}(j)] = D^{\mathsf{hb}}_{\mathsf{sec}}[j]$.

Figure 6: The Reduction $\mathcal{R}^{\mathcal{A}}$, pre-processing

can compute $\mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}^{\mathsf{hb}}_j]$ as $\mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}^{\mathsf{hb}}_j] = \mathbf{c}_i[g]$, 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 $(\texttt{COPY}, (\mathcal{BP}(g), \mathtt{loc}^{\mathsf{hb}}_j))$ by setting $D^{\mathsf{hb}}[\mathtt{loc}^{\mathsf{hb}}_j] \leftarrow D^{\mathsf{hb}}[\mathcal{BP}(g)]$, which results in the value that $(\mathbf{c}_1[g], \mathbf{c}_2[g])$ should have been an encoding of to be placed in $D^{\mathsf{hb}}[\mathtt{loc}^{\mathsf{hb}}_j]$, which is later read up by the CPU in Step 4c. In the hybrid model, this would result in the counter $\mathsf{C}[g]$ for that value to be incremented by one. Hence the leakage tally for each element stays below $\mathsf{C}[g] \cdot \mathsf{lb}_{\mathtt{bus}} \leq c \cdot \mathsf{lb}_{\mathtt{bus}}$.

- If the reply was $\perp$, then there is no way around computing $D_i = \mathsf{Dcon}_i(\mathbf{c}_i)$ (and then computing B_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 $\mathsf{lb}_{\mathtt{disk}} + c\mathsf{lb}_{\mathtt{bus}}$.

B Details of Our Hybrid Scheme

B.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 B.2 we will compile such a regular program into a hybrid RAM-scheme secure in the hybrid model (which can in turn

$\mathsf{same}^\star$ but also the index with which the tampered value matches.

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 $\texttt{CMD} = (\texttt{LEAK}, (\mathsf{Leak}_1, \mathsf{Leak}_2))$, then compute submit to $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_i, \cdot)$ the function $\mathsf{Leak}'_i = \mathsf{Leak}_i \circ \mathsf{Dcon}_i$. Let Λ be the value returned by the oracle; forward $(D^{\mathsf{hb}}_{\mathsf{pub}}, \Lambda)$ to $\mathcal{A}$.
3. If $\texttt{CMD} = (\texttt{TAMPER}, (\mathsf{Tamper}_1, \mathsf{Tamper}_2))$, virtually modify the disk $D_i \leftarrow \mathsf{Tamper}_i(D_i)$ by modifying the function Dcon_i as follows: $\mathsf{Dcon}_i \leftarrow \mathsf{Tamper}_i \circ \mathsf{Dcon}_i$.
4. If $\texttt{CMD} = (\texttt{EXEC}, (\mathsf{Leak}_1, \mathsf{Leak}_2), D')$ and $\mathtt{B} = 0$ then do the following:
   (a) Let $D^{\mathsf{sp}}_{\mathsf{pub}} \leftarrow D'$. If $\mathcal{E}^{-1}(D^{\mathsf{sp}}[\mathtt{pc}]) = \mathtt{sd}$, then self-destruct. Otherwise, compute $(\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}}) = \mathcal{E}^{-1}(D^{\mathsf{sp}}[\mathtt{pc}])$, and set $\mathtt{R}^{\mathsf{hb}}_0 = (\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}})$.
   (b) For $j = 1 \ldots, d$, let $\mathtt{loc}^{\mathsf{hb}}_j = \mathsf{I}^{\mathsf{hb}}[j]$. If $\mathtt{loc}^{\mathsf{hb}}_j < 2^{k-1}$, then let $\mathtt{R}^{\mathsf{hb}}_j = D^{\mathsf{hb}}[\mathtt{loc}^{\mathsf{hb}}_j]$. If $\mathtt{loc}^{\mathsf{hb}}_j \geq 2^{k-1}$, then submit the tampering query $(\mathsf{T}_1, \mathsf{T}_2)$, where $\mathsf{T}_i(\mathbf{c}_i) = \mathsf{Dcon}_i(\mathbf{c}_i)[\mathtt{loc}^{\mathsf{hb}}_j]$. If the reply is $(\mathsf{same}^\star, g)$, simulate the command $(\texttt{COPY}, (\mathcal{BP}(g), \mathtt{loc}^{\mathsf{hb}}_j))$ by setting $D^{\mathsf{hb}}[\mathtt{loc}^{\mathsf{hb}}_j] \leftarrow D^{\mathsf{hb}}[\mathcal{BP}(g)]$. If the reply is $\perp$, then simulate a self-destruct. Otherwise, compute $v = \mathsf{Decode}(\mathtt{crs}, v_1, v_2)$ and simulate the command $(\texttt{REPLACE}, (\mathtt{loc}^{\mathsf{hb}}_j, v))$ by setting $D^{\mathsf{hb}}[\mathtt{loc}^{\mathsf{hb}}_j] \leftarrow v$.
   (c) Compute $(D^{\mathsf{hb}}_{\mathsf{pub}}, D^{\mathsf{hb}}_{\mathsf{sec}}, \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{CPU}^{\mathsf{hb}}(D^{\mathsf{hb}}_{\mathsf{pub}}, D^{\mathsf{hb}}_{\mathsf{sec}}, \mathtt{pc}, \mathtt{ac})$. Update $\mathtt{pc}$ and $\mathtt{ac}$ and in case the self-destruct flag is set, simulate a self-destruct. Let $D^{\mathsf{sp}}_{\mathsf{pub}} = \mathcal{E}(D^{\mathsf{hb}}_{\mathsf{pub}})$.
   (d) For $j = 1 \ldots, d$, let $\mathtt{loc}^{\mathsf{hb}}_j = \mathsf{O}^{\mathsf{hb}}[j]$. How we process each $\mathtt{loc}^{\mathsf{hb}}_j$ depends on whether the above execution was a *secret execution* or not.
   public If $\mathtt{loc}^{\mathsf{hb}}_j \geq 2^{k-1}$, then let $v = D^{\mathsf{hb}}[\mathtt{loc}^{\mathsf{hb}}_j]$, sample $(v_1, v_2) \leftarrow \mathsf{Encode}(\mathtt{crs}, v)$ and update Dcon_i to $\mathsf{Dcon}'_i = \mathsf{Dcon}_i$, except that $(\mathsf{Dcon}'_i(\mathbf{c}_i))[\mathtt{loc}^{\mathsf{hb}}_j] = v_i$.
   secret If $\mathtt{loc}^{\mathsf{hb}}_j \geq 2^{k-1}$, then issue $(0, D^{\mathsf{hb}}[\mathtt{loc}^{\mathsf{hb}}_j])$ to make $\textsc{Game}^{\mathsf{comp},q,\mathsf{lb}_{\mathsf{code}}}_{\mathcal{C},\mathcal{R}}(b)$ generate an encoding $(v_{1,g}, v_{2,g})$—assume this was the g-th such request. As a result $v_{i,g}$ is added to $\mathbf{c}_i[g]$. Define $\mathsf{Dcon}'_i = \mathsf{Dcon}_i$, except that $(\mathsf{Dcon}'_i(\mathbf{c}_i))[\mathtt{loc}^{\mathsf{hb}}_j] = \mathbf{c}_i[g]$. Let $\mathcal{BP}(g)$ be the backup location used by $\mathcal{B}$; simulate the command $(\texttt{COPY}, (\mathtt{loc}^{\mathsf{hb}}_j, \mathcal{BP}(g)))$ by setting $D^{\mathsf{hb}}[\mathcal{BP}(g)] \leftarrow D^{\mathsf{hb}}[\mathtt{loc}^{\mathsf{hb}}_j]$.
   (e) Finally simulate the leakage $\Lambda_1 = \mathsf{Leak}_1(B_1)$ and $\Lambda_2 = \mathsf{Leak}_2(B_2)$ by computing $\mathsf{Bs} = (B_1, B_2)$ as in a real execution. If the execution was public the reduction knows all the values needed for doing this. If the execution was secret, the leakage will be computed with the aid of the leakage oracles. In particular, for $i = 1, 2$, submit to $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(\mathbf{c}_i, \cdot)$ the function Leak'_i which first computes the disk $D_i = \mathsf{Dcon}_i(\mathbf{c}_i)$ and then adds to B_i each value $v_i = D_i[\mathtt{loc}^{\mathsf{hb}}_j]$. Next, the function computes $D'_i = \mathsf{Dcon}'_i(\mathbf{c}_i)$ and adds each $v_i = D_i[\mathtt{loc}^{\mathsf{hb}}_j]$ from the writing phase to B_i. Hence, it returns $\Lambda_i = \mathsf{Leak}_i(B_i)$.

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

be transformed into a secure RAM-scheme via the emulator of Section 5).

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 ℓ_R is the size of the RAM, K is the position in the RAM where the key is stored, ℓ_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, ℓ_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, ℓ_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, ℓ_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 γ, 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 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 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 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 than the output itself. 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 values.

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

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

$$\omega^{\mathsf{hb}}_{\mathsf{pub}}[\mathtt{pc}] := (\mathtt{load\_input}, (2^{k-1}, I + i), (2^{k-1}, 2^{k-1} + I + i)\ .$$

The purpose of this instruction is to move the i-th word of the input from the public disk to the secret disk. Note that the instruction reads at $D^{\mathsf{hb}}[2^{k-1}] = \omega^{\mathsf{hb}}_{\mathsf{sec}}[0]$ 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

$$\omega^{\mathsf{hb}}_{\mathsf{pub}}[\mathtt{pc}] := (\mathtt{lift\_key}, (2^{k-1}, 2^{k-1} + K + i), (2^{k-1}, 2^{k-1} + K + i)\ .$$

The purpose of this instruction is to increment the activation value associated to 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

$$\omega^{\mathsf{hb}}_{\mathsf{pub}}[\mathtt{pc}] := (G_i, (2^{k-1}, 2^{k-1} + a_i, 2^{k-1} + b_i, 2^{k-1} + j), (2^{k-1}, 2^{k-1} + c_i, 2^{k-1} + j)\ ,$$

where $j = G + i$. The purpose of this instruction is to execute instruction number i. Note that the instruction reads at $D^{\mathsf{hb}}[2^{k-1} + G + i] = \omega^{\mathsf{hb}}_{\mathsf{sec}}[G + i]$, 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

$$\omega^{\mathsf{hb}}_{\mathsf{pub}}[\mathtt{pc}] := (\mathtt{reveal\_output}, (2^{k-1}, 2^{k-1} + i), (2^{k-1}, i)\ .$$

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

$$\omega^{\mathsf{hb}}_{\mathsf{pub}}[\mathtt{pc}] := (\mathtt{done}, 2^{k-1}, 2^{k-1})\ .$$

This is a sentinel instruction.

Figure 8: The Compiler, Public Disk.

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

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

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

$$\omega^{\mathsf{hb}}_{\mathsf{sec}}[j] := (2^{k-1} + j, L, -1, \ell_I + i, \mathsf{K}_i) \ .$$

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

$$\omega^{\mathsf{hb}}_{\mathsf{sec}}[j] := (2^{k-1} + j, L, -1, \ell_I + \ell_K + i, (G_i, a_i, b_i, c_i)) \ .$$

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

Figure 9: The Compiler, Secret Disk.

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 ι_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 α such that in the execution game, the RAM never reads a given position more than α 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.

B.2 The Compiled Program

The main idea behind the compiler is to store the program and all the intermediary values on the secret disk.[13] 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., $\omega^{\mathsf{hb}}_{\mathsf{sec}}[j] = (j, L, a, p, V)$, 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

[13] 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.

$\mathsf{Y}^{\mathsf{hb}} = \texttt{load\_input}$: If not $(\mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}}) = ((2^{k-1}, j), (2^{k-1}, 2^{k-1} + j))$, for some j, then self-destruct. Try to parse the input values as follows:

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

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

- $0 \leq \mathtt{pc} < \ell_I$ *// 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

$$\mathtt{O}_1^{\mathsf{hb}} \leftarrow (0, L, \mathtt{ac}, \mathtt{pc}, (K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)) \quad \textit{// write label back to } D_1^{\mathsf{hb}}$$
$$\mathtt{O}_2^{\mathsf{hb}} \leftarrow (j, L, \mathtt{ac}, \mathtt{pc}, x) \quad \textit{// store input in augmented form}$$

$\mathsf{Y}^{\mathsf{hb}} = \texttt{lift\_key}$: If not $(\mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}}) = (2^{k-1}, 2^{k-1} + j), (2^{k-1}, 2^{k-1} + j)$, for some j, then self-destruct. Try to parse the input values as follows

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

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

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

Otherwise, set

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

Figure 10: $\mathsf{Compute}^{\mathsf{hb}}$, Part I.

$\mathsf{Y}^{\mathsf{hb}} = \mathrm{G} \notin \{\texttt{load\_input}, \texttt{lift\_key}, \texttt{reveal\_output}, \texttt{done}\}$: If not $(\mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}}) = ((2^{k-1}, 2^{k-1} + a_i, 2^{k-1} + b_i, 2^{k-1} + j), (2^{k-1}, 2^{k-1} + c_i), 2^{k-1} + j))$, for some a_i, b_i, j, c_i, then self-destruct. Try to parse the input values as follows:

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

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 $D_{\mathsf{pub}}^{\mathsf{hb}}$ matches the one read from $D_{\mathsf{sec}}^{\mathsf{hb}}$*

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

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

Figure 11: $\mathsf{Compute}^{\mathsf{hb}}$, Part II.

$\mathsf{Y}^{\mathsf{hb}} = \texttt{reveal\_output}$: If not $(\mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}}) = ((2^{k-1}, 2^{k-1}+j), (2^{k-1}, j))$, for some j, then self-destruct. Try to parse the input values as follows:

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

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

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

$\mathsf{Y}^{\mathsf{hb}} = \texttt{done}$: If not $(\mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}}) = (2^{k-1}, 2^{k-1})$, for some j, then self-destruct. Try to parse the input values as follows:

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

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

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

and end the activation.

Figure 12: $\mathsf{Compute}^{\mathsf{hb}}$, Part III

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 $\omega^{\mathsf{hb}}_{\mathsf{sec}}[j]$. The above ensures that each value V occurring at $\omega^{\mathsf{hb}}_{\mathsf{sec}}[j] = (j, L, a, p, V)$ is the correct value V for memory position j and the current activation and program step. This property is maintained inductively.

The compiler $\mathbf{C}^{\mathsf{hb}}$ is given in Fig. 8–9. Recall that each instruction has a type $(\mathsf{Y}^{\mathsf{hb}}, \mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}})$; we call $(\mathsf{I}^{\mathsf{hb}}, \mathsf{O}^{\mathsf{hb}})$ the *IO pattern* of the instruction. Algorithm $\mathsf{Compute}^{\mathsf{hb}}$ is given in Fig. 10–12. As our hybrid CPU is deterministic, $\mathsf{Random}^{\mathsf{hb}}$ 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*.
- **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 instruction 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 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 $\mathsf{Compute}^{\mathsf{hb}}$, 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.

B.3 Analysis

We imagine the values in the secret disks being partitioned into so-called *domains*, labeled by $\delta \in \{0,1\}^\kappa$. If a value of the form $(j, \delta, \ldots)$ is stored into the secret disk, then we say that it is stored in the domain δ. 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 $\mathsf{CPU}^{\mathsf{hb}}$. One can easily verifies that these properties hold by the construction of $\mathsf{Compute}^{\mathsf{hb}}$ (c.f. Fig. 10–12):

1. If in a single execution the values read from the secret disk $D^{\mathsf{hb}}_{\mathsf{sec}}$ 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 δ, then all the values written to the secret disk by the CPU in that activation are written to domain δ.
3. The values that the CPU writes to the public disk $D^{\mathsf{hb}}_{\mathsf{pub}}$ do not depend on the domain the CPU reads from $D^{\mathsf{hb}}_{\mathsf{sec}}$. More precisely, changing *only* the domain δ in all the values read from the secret disk to some $\delta' \neq \delta$ will *not* change the value CPU writes into $D^{\mathsf{hb}}_{\mathsf{pub}}$.

We do a hybrid proof, starting from the real execution (in the hybrid model) and ending with the ideal execution. 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 $\mathcal{B} \in \mathbf{A}^{\mathsf{hb}}$ be any adversary attacking our hybrid RAM-scheme in the game $\textsc{Real}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B},\mathcal{G}}(k)$. 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 $\mathsf{CPU}^{\mathsf{hb}}$. More precisely, the adversary 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 $\mathsf{DESTRUCT}$ to be the event that the CPU self-destructs.

Lemma 6. *The probability of* $\mathsf{MIX} \wedge \neg\mathsf{DESTRUCT}$ *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 $\mathsf{DESTRUCT}$ did not occur the probability that MIX occurs in the next command and $\mathsf{DESTRUCT}$ does not is at most $2^{-\kappa+1}$. Then we prove this fact.

Assume we can prove that as long as MIX and $\mathsf{DESTRUCT}$ did not occur the probability that $\mathsf{MIX} \wedge \neg\mathsf{DESTRUCT}$ 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 $\mathsf{DESTRUCT}$ did not occur in an earlier step either. So, in all earlier steps, MIX and $\mathsf{DESTRUCT}$ did not occur. Hence, we know that in the current step the probability that $\mathsf{MIX} \wedge \neg\mathsf{DESTRUCT}$ occurs is at most $2^{-\kappa+1}$. However, by assumption, we know that MIX occurs, so either $\mathsf{MIX} \wedge \neg\mathsf{DESTRUCT}$ or $\mathsf{MIX} \wedge \mathsf{DESTRUCT}$ will occur. If $\mathsf{MIX} \wedge \mathsf{DESTRUCT}$ occurs, then there will be no more steps, as we self-destruct. Hence, the event $\mathsf{MIX} \wedge \neg\mathsf{DESTRUCT}$ 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 MIX and 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 MIX and 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 MIX occurs. We now analyse 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 $(j', L', \ldots)$; so this case occurs with probability at most g. After this event the average guessing probability is clearly 1, and can never 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 $\mathsf{RS}^{\mathsf{hb}}$, 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 $D^{\mathsf{hb}}_{\mathsf{sec}}$. In doing this, it keeps track of the record Q [14] defined within the hybrid model—it is easy to see that $\mathcal{B}$ can efficiently compute

[14] Recall that $\mathsf{Q}[i]$ is an identifier keeping track of whether the value at location i of the secret disk is known to the adversary or not.

1. Initially, let $\mathsf{Q}[i] = \top$ for all i, except that $\mathsf{Q}[0] = \bot$, and $\mathsf{Q}[j] = \bot$ for all key positions j and all code positions j. We use $\mathsf{Q}[i] = \top$ to represent $\mathsf{Q}[i] \neq \bot$, as we are not interested in the exact value of $\mathsf{Q}[i]$ when $\mathsf{Q}[i] \neq \bot$.
2. Receive $(P, \omega^{\mathsf{hb}}_{\mathsf{pub}})$ and Ω 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 $(\texttt{REPLACE}, (j, z))$, then update $\mathsf{V}[j] \leftarrow z$ and $\mathsf{Q}[j] \leftarrow \bot$.
5. When $\mathcal{B}$ outputs $(\texttt{COPY}, (j, j'))$, update $\mathsf{Q}[j'] \leftarrow \mathsf{Q}[j]$ and if $\mathsf{Q}[j] = \bot$, then $\mathsf{V}[j'] \leftarrow \mathsf{V}[j]$.
6. When $\mathcal{B}$ makes the command $(\texttt{EXEC}, D')$, then let $D^{\mathsf{hb}}_{\mathsf{pub}} \leftarrow D'$ be the modified public disk; inspect $D^{\mathsf{hb}}[\mathtt{pc}]$ to obtain the next instruction Suppose that, for the current execution, $(i_0, \ldots, i_g)$ are the positions to be read on the secret disk and $(j_0, \ldots, j_h)$ are the positions to be written on the secret disk.
   (a) If $\mathsf{Q}[i_0] = \cdots = \mathsf{Q}[i_g] = \bot$, which means all the values to be read are "known" (i.e., this is a public execution), then proceed as follows:
      - Use $\mathsf{V}[i_0], \ldots, \mathsf{V}[i_g]$ together with the values read from the public disk via the current instruction, $\mathtt{ac}$ and $\mathtt{pc}$ to compute the values to store into the output registers $\mathtt{O}^{\mathsf{hb}}_1, \ldots, \mathtt{O}^{\mathsf{hb}}_d$; note that these values are exactly the same values that $\mathsf{CPU}^{\mathsf{hb}}$ would have computed on $(\texttt{EXEC}, D')$.
      - Update Q by setting $\mathsf{Q}'[j_0] = \cdots = \mathsf{Q}'[j_h] := \bot$.
      - Similarly, update V as follows: for $a = 0, \ldots, h$, we set $\mathsf{V}[j_a]$ to hold the value that $\mathsf{CPU}^{\mathsf{hb}}$ would have written to $D^{\mathsf{hb}}_{\mathsf{sec}}[j_a]$, namely $\mathsf{V}[j_a] := \mathtt{O}^{\mathsf{hb}}_a$
      - Finally, for all positions in $D^{\mathsf{hb}}_{\mathsf{pub}}$ where $\mathsf{CPU}^{\mathsf{hb}}$ would have written, if any, update the public disk consistently.
   (b) Otherwise, this is a secret execution and we proceed as follows:
      i. If $\mathsf{Q}[i_0] \neq \bot, \ldots, \mathsf{Q}[i_g] \neq \bot$, then output $(\texttt{EXEC}, D')$. Then set $\mathsf{Q}[j_1] = \cdots = \mathsf{Q}[j_h] := \top$.
      ii. Else, there exists a and b such that $\mathsf{Q}[i_a] \neq \bot$ and $\mathsf{Q}[i_b] = \bot$. In that case, simulate a self-destruct (by ignoring this and all future $\texttt{EXEC}$ commands).

Figure 13: $\mathcal{B}_1$

an exact copy of Q. From the definition of Q, we observe that if $\mathsf{Q}[j] = \bot$, then the value in $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ is efficiently computable by $\mathcal{B}$. This holds because such value was either: (1) put in $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ by a $\texttt{REPLACE}$ command, (2) copied to $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ from $D^{\mathsf{hb}}_{\mathsf{sec}}[i]$, where $D^{\mathsf{hb}}_{\mathsf{sec}}[i]$ is efficiently computable, or (3) computed by the CPU by a run of $\mathsf{Compute}^{\mathsf{hb}}$ on inputs which were all efficiently computable by $\mathcal{B}$ (notice that our CPU is deterministic). It follows that $\mathcal{B}$ can efficiently compute $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ for all j where $\mathsf{Q}[j] = \bot$. For all j where $\mathsf{Q}[j] = \bot$ we will let $\mathcal{B}_1$ compute the value which would have been in $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$, had the $\texttt{REPLACE}$ commands of $\mathcal{B}$ been executed. Additionally, $\mathcal{B}_1$ will make sure that for all j where $\mathsf{Q}[j] \neq \bot$, it keeps in $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ the value that would have been in $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ if the $\texttt{REPLACE}$ commands of $\mathcal{B}$ had been executed. $\mathcal{B}_1$ maintains another record called V to store the known values—i.e., whenever it sets $\mathsf{Q}[j]$ to $\bot$, it updates $\mathsf{V}[j]$ accordingly. The details are given in Figure 13.

Consider the following event E, defined in both $\mathcal{B}$ and $\mathcal{B}_1$: there exist two positions j, j' in $D^{\mathsf{hb}}_{\mathsf{sec}}$ from which $\mathsf{CPU}^{\mathsf{hb}}$ reads in an execution, such that the $\mathsf{Q}[j] = \bot$ and $\mathsf{Q}[j'] \neq \bot$. Notice that,

for $\mathcal{B}_1$, the event E happens in Step 6(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 $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ where $\mathsf{Q}[j] \neq \perp$ that $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ is a value computed during the pre-processing (alternatively a value copied or computed from such values). That is, if $\mathsf{Q}[j] \neq \perp$ then $D^{\mathsf{hb}}_{\mathsf{sec}}[j]$ is in the secret domain L. On the other hand $\mathsf{Q}[j'] = \perp$ implies that the content of $D^{\mathsf{hb}}_{\mathsf{sec}}[j']$ is known to the adversary, and hence the value is in the public domain L' . Hence, when E occurs, we have that $\mathsf{CPU}^{\mathsf{hb}}$ reads from two domains L and L'. In such a case, by Property 1 of $\mathsf{CPU}^{\mathsf{hb}}$, $\mathsf{DESTRUCT}$ occurs if $L \neq L'$. On the other hand, if $L = L'$ the CPU does not self-destruct in $\text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B},\mathcal{G}}(k)$, and hence MIX occurs, which in turn implies that $\mathsf{MIX} \wedge \neg\mathsf{DESTRUCT}$ occurred. Using Lemma 6 (recall that $\kappa = k + 1$) we can conclude that:

$$\left\{\text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} \approx_{2^{-k}} \left\{\text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B}_1,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} .$$

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 $D^{\mathsf{hb}}_{\mathsf{sec}}$.*

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 have $a =$ `ac` and $p =$ `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 this happens in the same invocation of the CPU. In the following we use the requirements of the regular program (c.f. Section B.1). To conclude the proof, we considering a single execution of $\mathsf{CPU}^{\mathsf{hb}}$ in all possible cases:

1. `load_input`: $\mathsf{CPU}^{\mathsf{hb}}$ writes in locations 0 and j but we have $j \neq 0$ as $j \geq I > 0$ by *Reserve Nought*
2. `lift_key`: $\mathsf{CPU}^{\mathsf{hb}}$ writes in locations 0 and j but we have $j \neq 0$ as $j > 0$ by *Reserve Nought.*
3. `compute` G: $\mathsf{CPU}^{\mathsf{hb}}$ 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 $\mathsf{RS}^{\mathsf{hb}}$ 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 modifications to the public disk $D^{\mathsf{hb}}_{\mathsf{pub}}$ in positions j where $j \notin \{I, \ldots, I + \ell_I - 1\}$; note that such an adversary is not allowed to replace any value computed during pre-processing in $D^{\mathsf{hb}}_{\mathsf{pub}}$. Instead $\mathcal{B}_2$ keeps track of (i) which values would currently have been on the secret disk had the `COPY` command been performed, and (ii) the state of the public disk had the modification been applied. Towards this, $\mathcal{B}_2$ maintains a record $\widetilde{\omega}^{\mathsf{hb}}_{\mathsf{pub}}$ corresponding to $\omega^{\mathsf{hb}}_{\mathsf{pub}}$ (c.f. Fig. 4). When $\mathcal{B}_1$ makes an `EXEC` command, then $\mathcal{B}_2$ checks two things: (i) whether there is *any* replacement of the values put during pre-processing in $D^{\mathsf{hb}}_{\mathsf{pub}}$, i.e., if $\widetilde{\omega}^{\mathsf{hb}}_{\mathsf{pub}} \neq \omega^{\mathsf{hb}}_{\mathsf{pub}}$; (ii) whether the current location to be read from $D^{\mathsf{hb}}_{\mathsf{sec}}$ is copied from another position, i.e., the position or the counters in the augmented encoding fails to match, and `COPY` was already executed. If any of the above two checks fails, $\mathcal{B}_2$ simulates a self-destruct by ignoring all future `EXEC` commands. Otherwise $\mathcal{B}_2$ also outputs an `EXEC` command.

Lemma 8. *The following holds:*

$$\left\{\text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B}_1,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} = \left\{\text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B}_2,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}}.$$

Proof. By 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 $\omega^{\mathsf{hb}}_{\mathsf{pub}}[\mathtt{pc}]$ and the correct tuple (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 $\text{REAL}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B}_1,\mathcal{G}}(k)$ would have resulted in reading up an incorrect value from the public disk or an ill-formed tuple (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 an encoding 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 $\mathtt{ac}$, $\mathtt{pc}$, L and the (correct) parameters $(K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)$ retrieved form $\omega^{\mathsf{hb}}_{\mathsf{sec}}[0]$. For example, for $\mathtt{pc} \in \{K, \ldots, K+\ell_K-1\}$, the CPU can use K and ℓ_K to see that the label should be $\mathtt{lift\_key}$, which implies that the correct instruction should be $(\mathtt{lift\_key}, (2^{k-1}, 2^{k-1}+j), (2^{k-1}, 2^{k-1}+j))$ for $j = K+\mathtt{pc}-\ell_I$. From this the CPU 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 $\mathtt{pc} \in \{G, \ldots, G+\ell_G-1\}$ we have that, in a manner similar as above, the values j and L can be deduced from $\mathtt{ac}$, $\mathtt{pc}$ and $(K, \ell_K, I, \ell_I, G, \ell_G, O, \ell_O)$; similarly the position j and the expected tuple $(j, L, \mathtt{ac}-1, \mathtt{pc}, \ldots)$ can be deduced from $\mathtt{ac}$, $\mathtt{pc}$, 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 $H = (G_i, a_i, b_i, c_i)$. This implies that the check $H = (\mathrm{G}, a_i, b_i, c_i)$ guarantees that the instruction on the public disk was correct. The latter 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 $\mathtt{ac}$, 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 chooses values for the input positions namely $\{I, \ldots, I+\ell_I-1\}$ on the public disk. Consider the following simulator $\mathcal{S}_2$:

1. Initialize $\omega^{\mathsf{hb}}_{\mathsf{pub}}$ as the preprocessing would have done.
2. Run $\mathcal{B}_2$ until it executed command $(\mathtt{EXEC}, D')$ exactly ℓ_K times. Simulate by doing nothing.
3. For the next ℓ_I commands $(\mathtt{EXEC}, D')$ executed by D_2, simulate the modified public disk by setting $\omega^{\mathsf{hb}}_{\mathsf{pub}}[I+i] \leftarrow x_i$ for all $i = 0, \ldots, \ell_I - 1$.
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 $(\mathtt{EXEC}, D')$ exactly ℓ_G times. Simulate by doing nothing.
6. Compute $(y_0, \ldots, y_{\ell_O-1}) = \mathcal{Y}^{-1}((x_0, \ldots, x_{\ell_I-1}), y)$.[15]

[15]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.

7. Let $i = 0$. As long as $i < \ell_O$, run $\mathcal{B}_2$ to make it give the command (`EXEC`). In response to this update $\omega^{\mathsf{hb}}_{\mathsf{pub}}[O + i] \leftarrow y_i$, and let $i \leftarrow i + 1$.
8. Run $\mathcal{B}_2$ until it executed the command (`EXEC`) once. Then restart from Step 2.

It follows from *Output Simulatability* that

$$\left\{\textsc{Real}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B}_2,\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} = \{\textsc{Ideal}_{\mathcal{S}_2,\mathcal{G}}(k)\}_{k\in\mathbb{N}} \ .$$

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

$$\left\{\textsc{Real}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B},\mathcal{G}}(k)\right\}_{k\in\mathbb{N}} \approx_{2^{-k}} \{\textsc{Ideal}_{\mathcal{S},\mathcal{G}}(k)\}_{k\in\mathbb{N}} \ .$$

This show that the hybrid RAM-scheme $\mathsf{RS}^{\mathsf{hb}}$ is secure. Observe that because $\mathsf{RS}^{\mathsf{hb}}$ self-destructs the first time it reads up a value from the secret domain which is not the correct value, and since the correct values are touched at most α times by *Constant Fan-Out*, 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 $\mathcal{B}_2$. In $\textsc{Real}_{\mathsf{RS}^{\mathsf{hb}},\mathcal{B}_2,\mathcal{G}}(k)$ all values at input positions are written once and then read at most α 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 α times. This concludes the proof of Theorem 5.