A Tamper and Leakage Resilient von Neumann Architecture

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

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

Abstract. We present a *universal framework* for tamper and leakage resilient computation on a random access machine (RAM). 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.

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

* Received funding from the Marie Curie IEF/FP7 project GAPS, grant number: 626467.
** Partially supported by Danish Council for Independent Research via DFF Starting Grant 10-081612. Partially supported by the European Research Commission Starting Grant 279447.

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 [24, 23]. A compiler takes as input a description of some arbitrary cryptographic functionality $\mathcal{G}_{\mathsf{K}}$ and outputs a transformed functionality $\mathcal{G}'_{\mathsf{K}'}$ which has the same input/output 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 [23, 20, 10, 26, 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, 27, 15, 1, 6, 7, 17, 19, 2, 8, 9]), 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., [20]) 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. [27, 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, 27, 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, 27, 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

[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.

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 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

[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.

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.

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

[4] In [8] 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.

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., [25, 27] 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, 25, 30, 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 [24, 21, 14, 22, 29, 28]. While most works on leakage resilient circuit compilers require leakage-proof hardware, the breakthrough work of Goldwasser and Rothblum [22] shows how to completely eliminate leak-proof hardware for leakage in the split-state setting. It is an interesting open question, if one can use the compiler of [22] to implement our CPU and allow leakage also from its execution. We emphasize that most of the work on leakage resilient circuit compilers does not consider tampering attacks.

The concept of tamper resilient circuits has been introduced by Ishai, Prabhakaran, Sahai and Wagner [23] and further studied in [23, 20, 10, 26, 11]. On the upside such compilers require simpler 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 [26]) independently. That is, the adversary can set the bit carried on a wire to 1, set it to 0 or toggle its value. Moreover, it is assumed that in each execution at least a constant fraction of

[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.

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 $X_1 \approx_c X_2$ to denote that X_1 and X_2 are computationally indistinguishable meaning that for all PPT algorithms $\mathcal{A}$ we have that $\Pr[\mathcal{A}(X_1) = 1] - \Pr[\mathcal{A}(X_2) = 1] \leq negl(k)$.

[6] In [23, 20] 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 $(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1)) = (c_0, c_1)$.

The above one-shot game has been extended to the continuous setting in [17], where the adversary may tamper continuously with the encoding. In contrast to the above game, the adversary here obtains access to a tampering oracle $\mathcal{O}^q_{\mathsf{cnm}}((c_0, c_1), \cdot)$, where (c_0, c_1) is an encoding of either x_0 or x_1. The oracle can be queried up to q times with input functions $\mathsf{T}_0, \mathsf{T}_1 : \{0,1\}^n \to \{0,1\}^n$ and returns either $\mathsf{same}^\star$ (in case $(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1)) = (c_0, c_1)$), or $\perp$ (in case $\mathsf{Decode}(\mathtt{crs}, (\mathsf{T}_0(c_0), \mathsf{T}_1(c_1))) = \perp$), or $(\mathsf{T}_0(c_0), \mathsf{T}_1(c_1))$ in all other cases. The only additional restriction is that whenever $\perp$ is returned the oracle answers all further queries with $\perp$ (a.k.a. "self-destruct"). Furthermore, in the construction of [17] the adversary has access to leakage oracles $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_0, \cdot)$, $\mathcal{O}^{\mathsf{lb}_{\mathsf{code}}}(c_1, \cdot)$, that can be queried to retrieve up to $\mathsf{lb}_{\mathsf{code}}$ bits of information on each half of the target encoding. The access to the leakage oracles will be useful in our setting to obtain continuous leakage resilience on the buses. We refer the reader to the full version of this paper [18] for a precise definition of continuous non-malleable leakage resilient (CNMLR) codes.

Composability. We also introduce a notion of *adaptive composability* for CNMLR codes, where the adversary can specify two vectors of messages $\mathbf{x}_0 = (x_0^1, \ldots, x_0^m)$ and $\mathbf{x}_1 = (x_1^1, \ldots, x_1^m)$ (such that $|x_0^i| = |x_1^i|$) and the oracle $\mathcal{O}^q_{\mathsf{cnm}}(\mathbf{c}, \cdot)$ is parametrized by a vector of encodings $\mathbf{c} = (\mathbf{c}_0, \mathbf{c}_1) = ((c_0^1, \ldots, c_0^m), (c_1^1, \ldots, c_1^m))$ corresponding to either $\mathbf{x}_0$ or $\mathbf{x}_1$ (depending on the secret bit b above). The tampering functions now have a type $\mathsf{T}_0, \mathsf{T}_1 : (\{0,1\}^n)^m \to \{0,1\}^n$, and the oracle returns $(\mathsf{same}^\star, i)$ in case $c' = (c_0^i, c_1^i)$ for some $i \in [m]$. The leakage oracles are also parametrized by $\mathbf{c}_0$ and $\mathbf{c}_1$ and the adversary can leak up to $\mathsf{lb}_{\mathsf{code}}$ bits from each codeword.

Roughly a CNMLR code is adaptively m-composable if no PPT adversary can guess the value of b with noticeable advantage, even in case the messages in the two vectors $\mathbf{x}_0$, $\mathbf{x}_1$ are not fixed at the beginning of the game, but instead can be chosen adaptively when the game proceeds. A formal definition, together with a proof of the following theorem can be found in the full version.

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* $m = poly(k)$.

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[7] 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

[7] 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: $(\mathtt{crs}, D, \mathsf{pc}, \mathsf{ac}, \mathsf{Leak}_{\mathsf{Bs}})$
  // Loading...
Parse $D[\mathsf{pc}]$ 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: $(D, \mathtt{B}, \mathtt{T}, \lambda_{\mathsf{Bs}})$
```

Fig. 1. Algorithm CPU

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 $\mathcal{G}$ 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 1 (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* $(D, \mathtt{B}, \mathtt{T}) \leftarrow \mathsf{CPU}(\mathtt{crs}, D, \mathtt{pc}, \mathtt{ac})$.[8]
  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

[8] 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.

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

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 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 2 (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 3 (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 $\mathsf{lb}_{\mathsf{bus}}$ and we require that the length of the leakage returned by each function $\mathsf{Leak}_i^{\mathsf{sp}}$ is less than $\mathsf{lb}_{\mathsf{bus}}$.

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 $\mathsf{lb}_{\mathsf{bus}}$ we will denote the adversary class for a split-state RAM simply by $\mathbf{A}^{\mathsf{sp}} = (\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathsf{bus}})$. 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 $(\mathsf{lb}_{\mathsf{code}}, q)$-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 $(\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathsf{bus}})$-secure whenever $\mathsf{lb}_{\mathtt{disk}} + (c+1)\mathsf{lb}_{\mathsf{bus}} \leq \mathsf{lb}_{\mathsf{code}}$.*

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 $\mathsf{Tamper} = (\mathtt{COPY}, (j, j'))$ for $j, j' \geq 2^{k-1}$, then update $D[j'] \leftarrow D[j]$.
  – If $\mathsf{Tamper} = (\mathtt{REPLACE}, (j, \mathsf{val}))$ 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 $(\mathtt{COPY}, (j, j'))$, then let $\mathsf{Q}[j'] = \mathsf{Q}[j]$. When the adversary executes $(\mathtt{REPLACE}, (j, \mathsf{val}))$, then let $\mathsf{Q}[j] = \perp$. 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 $\mathsf{Q}[i_p] \neq \perp$, let $\mathsf{C}[\mathsf{Q}[i_p]] \leftarrow \mathsf{C}[\mathsf{Q}[i_p]] + 1$. Then proceed as follows. If $\mathsf{Q}[i_0] = \mathsf{Q}[i_1] = \cdots = \mathsf{Q}[i_d] = \perp$, then let

$\mathsf{Q}[j_1] = \cdots = \mathsf{Q}[j_d] = \perp$. 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 $\mathsf{Q}[j_i] \leftarrow \perp$.

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 $(\mathsf{lb}_{\mathsf{code}}, q)$-CNMLR code. Let $\mathbf{A}^{\mathsf{sp}} = (\mathsf{lb}_{\mathtt{disk}}, \mathsf{lb}_{\mathsf{bus}})$ be a split-state adversary class such that $\mathsf{lb}_{\mathtt{disk}} + (c+1) \cdot \mathsf{lb}_{\mathsf{bus}} \leq \mathsf{lb}_{\mathsf{code}}$. 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 the full version [18]; 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 $\omega^{\mathsf{sp}}_{\mathsf{pub}}$ be $\omega^{\mathsf{hb}}_{\mathsf{pub}}$. Then for each $j \in [0, |\omega^{\mathsf{hb}}_{\mathsf{sec}}|]$, let $\omega^{\mathsf{sp}}_{\mathsf{sec}}[j] = (\omega^{\mathsf{sp}}_{\mathsf{sec},1}[j], \omega^{\mathsf{sp}}_{\mathsf{sec},2}[j])$ be an encoding of $\omega^{\mathsf{hb}}_{\mathsf{sec}}[j]$ (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 $D^{\mathsf{sp}}_{\mathsf{sec}}$ and writes them back to disk. Then values v_j to be stored on $D^{\mathsf{sp}}_{\mathsf{pub}}$ 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 `REPLACE` and `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 `crs` and creates encodings of 0 for all the values on the secret disk using the CNMLR code, and puts dummy encodings $(v_1, v_2) \leftarrow \mathsf{Encode}(\mathtt{crs}, 0)$ 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 $\omega^{\mathsf{hb}}_{\mathsf{pub}}$ from $D^{\mathsf{hb}}_{\mathsf{pub}}$ and $\omega^{\mathsf{hb}}_{\mathsf{pub}} = \omega^{\mathsf{sp}}_{\mathsf{pub}}$. Depending on the queries in the online phase $\mathcal{B}$ will update these virtual disks in the following. `TAMPER` queries are simulated easily by applying the corresponding tamper functions to the current state of the virtual disks 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 `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 the full version [18] 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 the full version 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 $(\mathsf{ac}, \mathsf{pc})$ when the value was written. Intuitively, the secret label ensures that the adversary cannot use the "replace" command as that would require to guess the value of the label. On the other hand the position j will allow the CPU to check that it loaded a value from the right position, preventing the adversary to use the "copy" command to move values created by the CPU (or at setup) to another location. Finally, the pair (a, p) prevents the adversary from swapping values sharing the same L and the same j (i.e., 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 the full version of this paper.

Analysis. Next, we turn to a high-level overview of the security proof (the actual proof can be found in the full version). 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. D. Aggarwal, Y. Dodis, and S. Lovett. Non-malleable codes from additive combinatorics. *IACR Cryptology ePrint Archive*, 2013:201, 2013.
2. S. Agrawal, D. Gupta, H. K. Maji, O. Pandey, and M. Prabhakaran. Explicit non-malleable codes resistant to permutations and perturbations. *IACR Cryptology ePrint Archive*, 2014:316, 2014.
3. M. Bellare and D. Cash. Pseudorandom functions and permutations provably secure against related-key attacks. In *CRYPTO*, pages 666–684, 2010.
4. M. Bellare and T. Kohno. A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In *EUROCRYPT*, pages 491–506, 2003.
5. M. Bellare, K. G. Paterson, and S. Thomson. RKA security beyond the linear barrier: Ibe, encryption and signatures. In *ASIACRYPT*, pages 331–348, 2012.
6. M. Cheraghchi and V. Guruswami. Capacity of non-malleable codes. In *ICS*, pages 155–168, 2014.

7. M. Cheraghchi and V. Guruswami. Non-malleable coding against bit-wise and split-state tampering. In *TCC*, pages 440–464, 2014.
8. S. Coretti, U. Maurer, B. Tackmann, and D. Venturi. From single-bit to multi-bit public-key encryption via non-malleable codes. In *TCC*, 2015. To appear.
9. S. Coretti, Y. Dodis, B. Tackmann, and D. Venturi. Self-destruct non-malleability. *IACR Cryptology ePrint Archive*, 2014:866, 2014.
10. D. Dachman-Soled and Y. T. Kalai. Securing circuits against constant-rate tampering. In *CRYPTO*, pages 533–551, 2012.
11. D. Dachman-Soled and Y. T. Kalai. Securing circuits and protocols against 1/poly(k) tampering rate. In *TCC*, pages 540–565, 2014.
12. D. Dachman-Soled, F.-H. Liu, E. Shi, and H.-S. Zhou. Locally decodable and updatable non-malleable codes and their applications. In *TCC*, 2015. To appear.
13. I. Damgård, S. Faust, P. Mukherjee, and D. Venturi. Bounded tamper resilience: How to go beyond the algebraic barrier. In *ASIACRYPT (2)*, pages 140–160, 2013.
14. S. Dziembowski and S. Faust. Leakage-resilient circuits without computational assumptions. In *TCC*, pages 230–247, 2012.
15. S. Dziembowski, T. Kazana, and M. Obremski. Non-malleable codes from two-source extractors. In *CRYPTO (2)*, pages 239–257, 2013.
16. S. Dziembowski, K. Pietrzak, and D. Wichs. Non-malleable codes. In *ICS*, pages 434–452, 2010.
17. S. Faust, P. Mukherjee, J. B. Nielsen, and D. Venturi. Continuous non-malleable codes. In *TCC*, pages 465–488, 2014.
18. S. Faust, P. Mukherjee, J. B. Nielsen, and D. Venturi. A tamper and leakage resilient von Neumann architecture. Cryptology ePrint Archive, Report 2014/338, 2014. `http://eprint.iacr.org/`.
19. S. Faust, P. Mukherjee, D. Venturi, and D. Wichs. Efficient non-malleable codes and key-derivation for poly-size tampering circuits. In *EUROCRYPT*, pages 111–128, 2014.
20. S. Faust, K. Pietrzak, and D. Venturi. Tamper-proof circuits: How to trade leakage for tamper-resilience. In *ICALP (1)*, pages 391–402, 2011.
21. S. Faust, T. Rabin, L. Reyzin, E. Tromer, and V. Vaikuntanathan. Protecting circuits from leakage: the computationally-bounded and noisy cases. In *EUROCRYPT*, pages 135–156, 2010.
22. S. Goldwasser and G. N. Rothblum. How to compute in the presence of leakage. In *FOCS*, pages 31–40, 2012.
23. Y. Ishai, M. Prabhakaran, A. Sahai, and D. Wagner. Private circuits II: Keeping secrets in tamperable circuits. In *EUROCRYPT*, pages 308–327, 2006.
24. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.
25. Y. T. Kalai, B. Kanukurthi, and A. Sahai. Cryptography with tamperable and leaky memory. In *CRYPTO*, pages 373–390, 2011.
26. A. Kiayias and Y. Tselekounis. Tamper resilient circuits: The adversary at the gates. In *ASIACRYPT (2)*, pages 161–180, 2013.
27. F.-H. Liu and A. Lysyanskaya. Tamper and leakage resilience in the split-state model. In *CRYPTO*, pages 517–532, 2012.
28. E. Miles and E. Viola. Shielding circuits with groups. In *STOC*, pages 251–260, 2013.
29. E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In *EUROCRYPT*, pages 142–159, 2013.
30. H. Wee. Public key encryption against related key attacks. In *Public Key Cryptography*, pages 262–279, 2012.