Securing Circuits Against Constant-Rate Tampering

Dana Dachman-Soled and Yael Tauman Kalai

Microsoft Research New England

Abstract. We present a compiler that converts any circuit into one that remains secure even if a *constant fraction* of its wires are tampered with. Following the seminal work of Ishai *et al.* (Eurocrypt 2006), we consider adversaries who may choose an arbitrary set of wires to corrupt, and may set each such wire to 0 or to 1, or may toggle with the wire. We prove that such adversaries, who *continuously* tamper with the circuit, can learn at most *logarithmically many bits* of secret information (in addition to black-box access to the circuit). Our results are information theoretic.

Key words: side-channel attacks, tampering, circuit compiler, PCP of proximity

1 Introduction

In recent years, there has been a proliferation of physical attacks on cryptographic schemes. Such attacks exploit the implementation (rather than the functionality) of the scheme. For instance, Kocher *et al.* [46] demonstrated how one can possibly learn the secret key of an encryption scheme by measuring the power consumed during an encryption operation, or by measuring the time it takes for the operation to complete [45]. Other types of physical attacks include: inducing faults to the computation [5, 7, 46], using electromagnetic radiation [30, 55, 54], and several others [54, 43, 47, 35]. These physical attacks have proven to be a significant threat to the practical security of cryptographic devices.

Traditional cryptographic models do not take such attacks into account since they idealize the parties' interaction and implicitly assume that an adversary may only observe an honest party's input-output behavior. Recently, a large and growing body of research has sought to introduce more realistic models and to secure cryptographic systems against such physical attacks. The vast majority of these works focus on securing cryptographic schemes against various *leakage* attacks (e.g. [11, 38, 49, 31, 37, 20, 51, 1, 50, 42, 18, 17, 24, 39, 34]). In these attacks an adversary plays a passive role, learning information about the honest party through side-channels but not attempting to interfere with the honest party's computation. However, as mentioned above, physical attacks are not limited to leakage, and include active *tampering* attacks, where an adversary may actively modify the honest party's memory or circuit. The focus of this work is to construct schemes that are secure even in the presence of tampering (we elaborate on related work in Section 1.4).

1.1 Our Results

We show a general compiler that converts any circuit into one that is resilient to (a certain form of) tampering. We consider the tampering model of Ishai *et al.* [37]. Namely, we consider *adversarial* faults, where during each run the computationally unbounded adversary can tamper with any (bounded) set of wires of his choice.

We note that we cannot hope to get correctness, since the adversary may simply tamper with the final output wire. Thus, following [37], we do not attempt to guarantee correctness, but instead devote our efforts to ensuring *privacy*. More specifically, we consider circuits that are associated with a secret state (such as a cryptographic key). We model such circuits as standard circuits (with AND, OR, and NOT gates), with additional secret memory that contains the secret state. The circuit itself is thought of as being public and known to the adversary, whereas the memory content is secret. Following the terminology of [37], we refer to such circuits as *private circuits*. Our goal is to protect the secret state of the circuit even when an adversary may run the circuit on arbitrary inputs while continuously tampering with the circuit.

Unlike the leakage regime, where we build cryptographic schemes that are secure against *arbitrary* (bounded) leakage, in the tampering regime, we focus on limited types of adversarial tampering attacks. Indeed, it is not hard to see that it is impossible to construct private circuits resilient to arbitrary tampering attacks, since an adversary may modify the circuit so that it simply outputs the entire secret state in memory. As in [37], we only allow the adversary to tamper with individual wires [37, 25] and individual memory gates [14, 31, 21]. More specifically, in each run of the circuit we allow the adversary to specify a set of tampering instructions, where each instruction is of the form: Set a wire (or a memory gate) to 0 or 1, or toggle with the value on a wire (or a memory gate). However, in contrast to [37], we allow the adversary to tamper with any *constant fraction* of wires and memory gates in the circuit. We note that the tampering allowed in [37] is very local: To be resilient against tampering with t wires per run, they blow up the size of the circuit by a factor of at least t; thus their overall tampering rate is very small compared to the size of the circuit.

As noted by [31], it is impossible to construct private circuits resilient against tampering on wires without allowing feedback into memory, i.e. without allowing the circuit to overwrite its own memory. Otherwise, an adversary may simply set to 0 or 1 one memory gate at a time and observe whether the final output is modified or not. Thus, the adversary may often learn the entire internal state of the circuit by setting one wire at a time.

Even if we allow feedback, and place limitations on the type of tampering we allow, it is not a priori clear how to build tamper-resilient circuits. As pointed out in [37], the fundamental problem is that the part of the circuit which is supposed to detect tampering and overwrite the memory, may itself be tampered with. Indeed, this "self-destruct" mechanism itself needs to be resilient to tampering.

As in [37], we prove security using a simulation based definition, where we require that for any adversary who continually tampers with the circuit (as described above), there exists a simulator who simulates the adversary's view. However, whereas [37] give the simulator only black-box access to the circuit, we also give the simulator logarithmic number of leakage bits on the secret state. We note that for many applications, leakage of only logarithmically many bits of the secret key does not break the security of the primitive. This is because an adversary may simply *guess* these bits of leakage on his own.

We also note that our compiler is deterministic, and for deterministic constructions such leakage is necessary. Loosely speaking, this is the case since an adversary can always leak a memory gate of its choice, by checking whether setting that memory gate to 0 affects the functionality of the circuit. We note that if the compiler is deterministic then many of the memory gates contain "sensitive" information. This is not necessarily the case if the compiler is randomized since then the secret state can be secret-shared in memory, in which case each memory gate on its own may contain a truly random bit.

Our Results More Formally We present a general compiler $\mathcal{T}$ that converts a circuit C with a secret state s (denoted by C_s) into a circuit $\tilde{C}_{\tilde{S}}$ with a secret state $\tilde{S}$. We consider computationally unbounded adversaries $\mathcal{A}$ who receive access to $\tilde{C}_{\tilde{S}}$ and behave in the following way: $\mathcal{A}$ runs the circuit many times with arbitrary and adaptively chosen inputs. In addition, during each run of the circuit the adversary $\mathcal{A}$ may specify tampering instructions of the form "set wire w to 1","set wire w to 0", "flip value of wire w", as well as "set memory gate g to 1", "set memory gate g to 0", "flip value of memory gate g", for any wire w or memory gate g. We restrict the number of tampering instructions $\mathcal{A}$ may specify per run to be at most $\lambda \cdot \sigma$, where $\lambda > 0$ is some constant and σ is the size of the circuit $\tilde{C}_{\tilde{S}}$. Thus, in each run, $\mathcal{A}$ may tamper with a constant fraction of wires and memory gates. Again, we emphasize that our result does not rely on computational assumptions and is completely information theoretic. Moreover, our result does not rely on any source of randomness and our compiler is deterministic.

Theorem 1 (Main Theorem, Informal). *There exists a universal constant $\lambda > 0$ and an efficient transformation $\mathcal{T}$ which takes as input any circuit C_s with private state s and outputs a circuit $\tilde{C}_{\tilde{S}}$ with private state $\tilde{S}$ such that the following two conditions hold:*

Correctness: For every input x, $C_s(x) = \tilde{C}_{\tilde{S}}(x)$.

Tamper-Resilience: For every adversary $\mathcal{A}$, which may tamper with a λ-fraction of wires and memory gates in $\tilde{C}_{\tilde{S}}$ per run, there exists a simulator Sim, *which can simulate the view of $\mathcal{A}$ given only black-box access to C_s and* $\log(T)$ *bits of leakage on s, where T is an upper bound on the number of times that $\mathcal{A}$ runs the (tampered) circuit.*[1] *Moreover, the runtime of* Sim *is polynomial in the runtime of $\mathcal{A}$ and C_s.*

Intuitively, the theorem asserts that adversaries who may observe the input-output behavior of the circuit while tampering with at most a λ-fraction of wires and memory gates in each run, do not get too much extra knowledge over

[1] The reader can think of T as polynomial in the security parameter.

what they could learn from just input-output access to the circuit. More specifically, running the (tampered) circuit $\text{poly}(|s|)$ times leaks at most $O(\log |s|)$ bits.

We remark that this relaxation of allowing $O(\log |s|)$ bits of leakage was first considered by Faust *et. al.* [25]. We additionally mention, as we argued previously, that leaking at most $O(\log |s|)$ bits does not compromise security of many applications. This is because an adversary receiving only input-output access to the circuit may simply *guess* the $O(\log |s|)$ bits of leakage, and his guess will be correct with probability $1/2^{O(\log |s|)} = 1/\text{poly}(|s|)$. Thus, we don't lose much by leaking $O(\log |s|)$ bits of the secret key.

1.2 Comparison with Ishai *et al.* [37]

Our work is inspired by the work of [37]. As in our work, they too consider circuits with memory gates, and consider the same type of faults as we do. Similarly to us, they construct a general compiler that converts any private circuit into a tamper resilient one. However our work differs from their work in several ways.

- The tamper resilient circuits in [37] require the use of "randomness gates", which output a fresh random bit in each run of the circuit. Alternatively, [37] can get rid of these randomness gates at the cost of relying on a computational assumption. Their computational result, which does not require randomness gates, relies on the existence of one-way functions and considers only polynomially bounded adversaries. In contrast, our compiler and the resulting tamper-resilient circuits are deterministic, and provide information theoretical security.
- As mentioned previously, [37] constructs tamper resilient circuits that are resilient only to local tampering. More specifically, in order to be resilient to tampering with t wires per run, they blow up the circuit size by a factor of at least t. In contrast, our tamper-resilient circuits are resilient to a constant fraction of tampering anywhere in the circuit, and thus are robust to global tampering.
- In the tamper resilient circuits of [37], there is one gate G that has a large fan-out. In practice, however, large fan-out gates do not exist, and instead they are implemented using a *splitter*, which takes as input the output wire of a gate and duplicates the wire many times. Unfortunately, such an implementation is not resilient to tampering with the output wire of G (which is the input wire to the splitter), since if this wire is tampered with, then it causes an immediate tampering with *all* the output wires of the splitter. In contrast, in our work all the gates of the tamper resilient circuits have constant fan-out (and some gates use splitters).
- One advantage of the tampering model of [37] over our model, is that their model allows for"persistent faults", e.g, if a value of some wire is fixed during one run, it remains set to that value in subsequent runs. We note that in our case, we cannot in general allow persistent faults across runs of the circuit. However, we allow "persistent faults" on memory gates so if a memory value is modified during one run, it remains modified for all subsequent runs.
- Another advantage of [37] is that their security definition is slightly stronger than ours. They guarantee that any adversary that continually tampers with the circuit (as described above) can be simulated given only black-box access to the circuit, whereas in our security definition, we give the simulator in addition logarithmically many bits of information.

Extending our result to allow for both Leakage and Tampering. Note that so far, we considered adversaries who only tamper with the wires and the memory of the circuit. Our result can be extended to consider adversaries who both tamper and leak. More specifically, we show a general compiler that converts any circuit into one that is resilient against both tampering (in the model described above) and leakage. The leakage model that we consider is OCL ("only computation leaks") leakage, as defined by Micali and Reyzin [49].

Unfortunately, being resilient to leakage (in addition to tampering) comes at a price: First, the resulting circuit no longer tolerates a constant-fraction of tampering, but rather security is ensured as long as $1/\text{poly}(k)$-fraction of the wires (or memory gates) can be tampered with. In addition, the result is no longer information theoretical, and relies on cryptographic assumptions. Specifically it relies on the existence of a non-committing encryption scheme [12] (which can be instantiated from hardness of factoring Blum integers, CDH and LWE [12, 15, 13]), and on the existence of a fully homomorphic encryption scheme (which can be instantiated from LWE [32, 10, 9]). We defer the reader to Section 6 for details.

1.3 Related Work

The problem of constructing error resilient circuits dates back to the work of Von Neumann from 1956 [56]. Von Neumann studied a model of *random* errors, where each gate has an (arbitrary) error independently with small fixed probability, and his goal was to obtain *correctness* (as opposed to privacy). There have been numerous follow up papers to this seminal work, including [16, 53, 52, 27, 23, 36, 28, 22], who considered the same noise model, ultimately showing that any circuit of size σ can be encoded into a circuit of size $O(\sigma \log \sigma)$ that tolerates a fixed constant noise rate, and that any such encoding must have size $\Omega(\sigma \log \sigma)$.

There has been little work on constructing circuits resilient to *adversarial* faults, while guaranteeing correctness. The main works in this arena are those of Kalai *et al.* [41], Kleitnam *et al.* [44], and Gál and Szegedy [29]. The works of [44] and [41] consider a different model where the only type of faults allowed are short-circuiting gates. [29] consider a model that allows arbitrary faults on gates, and show how to construct tamper-resilient circuits for symmetric Boolean functions. We note that [29] allow a constant fraction δ of adversarial faults *per level* of the circuit. Moreover, if there are less than $1/\delta$ gates on some level, they allow no tampering at all on that level. [29] also give a more general construction for any efficiently computable function which relies on PCP's. However, in order for their construction to work, they require an entire PCP proof π of correctness of the output to be precomputed and handed along with the input to the tamper-resilient circuit. Thus, they assume that the input to the circuit is already encoded via an encoding which depends on the *output* value of that very circuit. We also use the PCP methodology in our result, but do not require any precomputations or that the input be encoded in some special format.

Recently, the problem of physical attacks has come to the forefront in the cryptography community. From the viewpoint of cryptography, the main focus is no longer to ensure correctness, but to ensure *privacy*. Namely, we would like to protect the honest party's secret information from being compromised through the physical attacks of an adversary. There has been much work on protecting circuits against leakage attacks [38, 49, 20, 51, 19, 26, 39, 34]. However, there has not been much previous work on constructing circuits resilient to tampering attacks. In this arena, there have been two categories of works. The works of [31, 21, 14, 48, 40] allow the adversary to only tamper with and/or leak on the *memory* of the circuit in between runs of the circuit, but do not allow the adversary to tamper with the circuit itself. We elaborate on these related works in Section 1.4. We note that this model of allowing tampering only with memory is very similar to the problem of "related key attacks" (see [3, 2] and references therein). In contrast, in our work, as well as in the works of [37, 25], the focus is on constructing circuits resilient to tampering with both the memory as well as the wires of the circuit.

Faust *et al.* [25] consider a model that is reminiscent to the model of [37] and to the model we consider here. They consider adversarial faults where the adversary may actually tamper with all wires of the circuit but each tampering attack fails independently with some probability δ. As in our case, they also allow the adversary to learn a logarithmic number of bits of information on the secret key. However, their result requires the use of small tamper-proof hardware components.

1.4 Related work on Memory Tampering

In this section, we discuss related work in the model where only the memory may be tampered with [31, 21, 48, 40, 14].

Gennaro *et al.* [31] present a negative result, showing that without the use of secure hardware an attacker can recover the entire secret information stored in the memory (say, a signature key), by sequentially setting or resetting bits of the memory and observing the effects of these changes on the output. Due to this impossibility result, [31] turn to proving security in a model where devices have two separate components: one is tamper-proof yet public (circuitry), and the other is tamperable yet secret (memory). In contrast, our work (similarly to the work of Ishai *et al.* [37]) gets around the impossibility result by allowing to feed values back into the memory. As was noted in [37], this form of feedback, which prevails in real-world computing devices, is essential for realizing the strong notion of privacy considered in this work.

Dziembowski *et al.* [21] introduce the notion of non-malleable codes. Such codes have the property that, for some specific class of tampering functions, tampering with a codeword will either cause the decoding algorithm to output $\perp$, or output a codeword which is unrelated to the original encoding. They show how to construct non-malleable codes for certain classes of tampering functions. They also show that non-malleable codes are sufficient for achieving security in the model of [31].

Liu and Lysyanskaya [48] consider both tampering and probing attacks on memory. They show an impossibility result on proving security against adversaries who may both leak and tamper when the circuit does not have access to a source of random bits. Again, unlike our model, they consider only leakage and tampering with memory, and not with the wires of the circuit. Kalai *et al.* [40] also consider tampering and leakage attacks, and construct signature and encryption schemes that are secure against adversaries that (continually) tamper with and leak on the secret key.

Choi *et al.* [14] introduced the notion of Built-in Tamper Resilience, which is a security definition for protocols where some of the parties may be implemented by hardware tokens (and thus may be tampered with). They give some constructions of specific cryptographic protocols satisfying this definition and also prove a composition theorem for their definition. However, as in [31], their model allows only tampering with the memory, but not with the circuit itself.

1.5 Our Techniques

Our conceptual approach is similar to [37]: The tamper-resilient circuit has an "error-detection" mechanism, and if an error is detected then the circuit self-destructs (i.e. the memory is zeroed out). However, our techniques for achieving this are very different.

Recall that our goal is to guarantee security against an adversary who may tamper with a *constant fraction* of the wires and memory gates. We begin with the simple observation that circuits in NC^0 are inherently tamper-resilient. In particular, since each output bit of an NC^0 circuit is computed by a constant-size circuit (say of size $\tilde{\sigma}$), if we allow $\alpha \cdot 1/\tilde{\sigma}$-fraction of tampering for some constant $\alpha < 1$, then we ensure that at least $(1-\alpha)$-fraction of the output bits are exactly correct.

In order to construct a good error-detection mechanism for our tamper-resilient circuit, we need to construct a mechanism that detects errors and is itself resilient to a constant-fraction of tampering. More specifically, we would like to construct a "robust" verifier in NC^0 for the computation of the original circuit, such that if an error is detected in the computation, then most of the circuit's output are set to 0. Fortunately, the PCPP (PCP of Proximity) Theorem [4] provides us with exactly such a verifier. Informally, in a PCPP for an NP language $\mathcal{L}$, the verifier is given oracle access to a PCP π and is also given oracle access to an instance z. The verifier tosses logarithmically many random coins and it queries a *constant* number of bits of π and z. It accepts if $z \in \mathcal{L}$ and if the PCP π was generated honestly, and it rejects (with probability $1/2$) if z is far from being in the language (even if π was generated maliciously). We refer the reader to Section 2 for details.

Note that for any fixed setting of the random coins, the corresponding verifier is of constant size since it has only a constant number of input bits. Thus, by constructing a separate verifier corresponding to each possible outcome of the random coins and outputting the output bits of all verifiers, we obtain a single (larger) verifier in NC^0 with the property that if z is far from $\mathcal{L}$ then at least a $1/2$-fraction of the verifier's outputs are 0. Note that the soundness property only holds when z is "far" from every string in $\mathcal{L}$. Thus, our transformed circuit has the following basic paradigm: We encode the secret state s and the input x using an error-correcting code to obtain $S = \mathsf{ECC}(s)$ and $X = \mathsf{ECC}(x)$. We compute $b = C_s(x)$, and we compute a PCPP proof π that $(b, S \circ X) \in \mathcal{L}$ where $\mathcal{L} = \{(b, S \circ X) \mid \exists s, x : S = \mathsf{ECC}(s), X = \mathsf{ECC}(x), b = C_s(x)\}$. Then we verify the proof and self-destruct (i.e. erase all memory) if any of the output bits of the verifier are 0. Additional work is required to go from tolerating a constant-fraction of tampering in the error-detection stage to tolerating a constant-fraction of tampering overall. We elaborate on these in Section 4.2.

We mention that the resulting tamper-resilient circuit has one gate with large fan-out. Additional ideas are needed in order to remove the need of such a large fan-out gate. We elaborate on these in Section 5.

We note that the techniques mentioned so far are used to get resilience only against tampering attacks (and not leakage attacks). In order to get security against both (continual) leakage and tampering, we rely on techniques in the leakage regime, and in particular rely on recent results in the OCL model [49, 39, 34, 33].[2] We discuss the details in Section 6.

[2] We actually rely on the recently constructed LDS compiler [6], which in turn is based on OCL compilers.

2 PCP of Proximity

In this section, we present necessary preliminaries on PCP of proximities (denoted by PCPP), and state the efficient PCPP theorem of [4]. A PCP of proximity is a relaxation of a standard PCP, that only verifies that the input is *close* to an element of the language. The advantage of this relaxation is that it allows the possibility that the verifier may read only a small number of bits from the input. For greater generality, the input is divided into two parts (a, z), where a is given explicitly to the verifier, and z is given only as an oracle. Thus PCPs of proximity consider languages which consist of pairs of strings, and therefore are referred to as *pair languages*.

Definition 1 (Pair language). *A* pair language *L is simply a subset of the set of string pairs $L \subseteq \{0,1\}^* \times \{0,1\}^*$. For every $a \in \{0,1\}^*$, we denote $L_a = \{z \in \{0,1\}^* : (a,z) \in L\}$. We usually denote $\ell = |a|$ and $K = |z|$.*

Definition 2 (Relative Hamming distance). *Let $z, z' \in \{0,1\}^K$ be two strings. The* relative Hamming distance *between z and z' is defined as*

$$\Delta(z, z') \triangleq |\{i \in [K] : z_i \neq z'_i\}|/K.$$

*We say that z is δ-*far *from z' if $\Delta(z,z') \geq \delta$. More generally, let $S \subseteq \{0,1\}^K$; we say z is δ-*far *from S if $\Delta(z,z') \geq \delta$ for every $z' \in S$.*

Definition 3 (PCP **Verifiers).** *A* verifier *is a probabilistic polynomial time algorithm V that, on an input x, tosses $r = r(|x|)$ random coins and generates a sequence of $q = q(|x|)$ queries $I = (i_1, \ldots, i_q)$ and a circuit $D : \{0,1\}^q \to \{0,1\}$ of size at most $d(|x|)$.*[3]

We think of V as representing a probabilistic oracle machine that queries its oracle π at positions I, receives the q answer bits $\pi|_I \triangleq (\pi_{i_1}, \ldots, \pi_{i_q})$, and accepts if and only if $D(\pi_I) = 1$. We write $(I, D) \leftarrow V(x)$ to denote the queries and circuit generated by V on input x (and random coin tosses). We call r the randomness complexity, *q the* query complexity, *and d the* decision complexity *of V.*

Definition 4 (PCPP **verifier for a pair language [4]).** *For functions $s, \delta : \mathbb{N} \to [0,1]$, a verifier V is a* probabilistically checkable proof of proximity *(*PCPP*) system for a pair language L with* proximity parameter *δ and* soundness error *s, if the following two conditions hold for every pair of strings (a, z):*

Completeness: If $(a,z) \in L$ then there exists π such that $V(a)$ accepts oracle $z \circ \pi$ with probability 1. Formally

$$\exists \pi \Pr_{(I,D) \leftarrow V(a)} [D((z,\pi)|_I) = 1] = 1.$$

Soundness: If z is $\delta(|a|)$-far from $L(a)$, then for every π, the verifier $V(a)$ accepts oracle $z \circ \pi$ with probability strictly less than $s(|a|)$. Formally,

$$\forall \pi \Pr_{(I,D) \leftarrow V(a)} [D((z \circ \pi)|_I) = 1] < s(|a|).$$

Theorem 2 (Efficient PCPP **for Pair-language (Theorem 3.3 of [4])).** *Let $T : \mathcal{N} \to \mathcal{N}$ be a non-decreasing function, and let L be a pair language. If L can be decided in time T,*[4] *then for every constant $\rho \in (0,1)$ there exists a PCP of proximity for L with randomness complexity $O(\log T)$, query complexity $q = O(1/\rho)$, perfect completeness, soundness error $1/2$, and proximity parameter ρ.*

We mention that [4] does not discuss the complexity of constructing the PCPP proof, but the efficiency follows by a close inspection of their construction.

[3] For the sake of simplicity we consider only bit queries.

[4] L can be decided in time T if there exists a Turing machine M such that for every input $(a,b) \in \{0,1\}^* \times \{0,1\}^*$, $M(a,b) = 1$ if and only if $(a,b) \in L$, and $M(a,b)$ runs in time $T(|a| + |b|)$.

3 The Tampering Model

3.1 Circuits with Memory Gates

Similarly to [37], we consider a circuit model that includes memory gates. Namely, a circuit consists of (the usual) AND, OR, and NOT gates, connected to each other via wires, as well as input wires and output wires. In addition, a circuit may have memory gates. Each memory gate has one (or more) input wires and one (or more) output wires. Each memory gate is initialized with a bit value 0 or 1. This value can be updated during each run of the circuit.

Each time the circuit is run with some input x, all the wires obtain a $0/1$ value. The values of the input wires to the memory gates define the way the memory is updated. We allow only two types of updates: delete or unchange. Specifically, if an input wire to a memory gate has the value 0, then the memory gate is overwritten with the value 0. If an input wire to a memory gate has the value 1, then the value of the memory gate remains unchanged. We denote a circuit C initialized with memory s by C_s.

3.2 Tampering Attacks

We consider adversaries, that can carry out the following attack: The adversary has black-box access to the circuit, and thus can repeatedly run the circuit on inputs of his choice. Each time the adversary runs the circuit with some input x, he can tamper with the wires and the memory gates. We consider the following type of faults: Setting a wire (or a memory gate) to 0 or 1, or toggling with the value on a wire (or a memory gate).

More specifically, the adversary can adaptively choose an input x_i and a set of tampering instructions (as above), and he receives the output of the tampered circuit on input x_i. He can do this adaptively as many times as he wishes. We emphasize that once the memory has been updated, say from s to s', the adversary no longer has access to the original circuit C_s, and now only has access to $C_{s'}$. Namely, the memory errors are persistent, while the wire errors are not persistent.

We denote by $\mathsf{TAMP}_{\mathcal{A}}(C_s)$ the output of an adversary $\mathcal{A}$ that carries out the above tampering attack on a circuit C_s. We say that an adversary $\mathcal{A}$ is a λ-tampering adversary if during each run of the circuit he tampers with at most a λ-fraction of the circuit. Namely, $\mathcal{A}$ can make at most $\lambda \cdot |C_s|$ tampering instructions for each run, where each instruction corresponds either to a wire tampering or to a memory gate tampering.
Remark. In this work, we define the size of a circuit C, denoted by $|C|$, as the number of wires in C plus the number of memory gates in C. Note that this is not the common definition (where usually the size includes also the gates); however, it is equivalent to the common definition up to constant factors.

To define security of a circuit against tampering attacks we use a simulation-based definition, where we compare the real world, where an adversary $\mathcal{A}$ (repeatedly) tampers with a circuit C_s as above, to a simulated world, where a simulator Sim tries to simulate the output of $\mathcal{A}$, while given only black-box access to the circuit C_s, and without tampering with the circuit at all.

In this work, we give the simulator Sim, in addition to black box access to the circuit C_s, the privilege to request $\log T$ bits of information about s, where T is the number of times $\mathcal{A}$ runs the tampered circuit. More specifically, the simulator, in addition to black-box access to the circuit C_s, is also given oracle access to a leakage oracle that takes as input any leakage request, represented as a boolean circuit $L : \{0,1\}^k \to \{0,1\}$ and returns $L(s)$. The simulator Sim may query the leakage oracle at most $\log T$ times. We denote the output of Sim by $\mathsf{LeakBB}_{\mathsf{Sim},\log T}(C_s)$. As noted in the Introduction, this leakage is necessary if we restrict ourselves to deterministic constructions.

Definition 5. *We say that a circuit C_s is secure against λ-tampering adversaries, if for every λ-tampering adversary $\mathcal{A}$ there exists a simulator* Sim*, that runs in time* $\mathrm{poly}(|\mathcal{A}|, |C_s|)$*, such that*

$$\{\mathsf{TAMP}_{\mathcal{A}}(C_s)\}_{k\in\mathbb{N}} \equiv \{\mathsf{LeakBB}_{\mathsf{Sim},\log T}(C_s)\}_{k\in\mathbb{N}},$$

where T is an upper bound on the number of times that $\mathcal{A}$ runs the (tampered) circuit.

We give an efficient method for constructing circuits that remain secure against adversaries that tamper with a constant fraction of the wires in the circuit. Namely, we prove that there exists a constant $\lambda > 0$ such that the resulting circuit is secure against λ-tampering adversaries.

4 The Compiler

In this section, we present our efficient compiler $\mathcal{T}$, which takes a circuit C_s, with memory containing a secret $s \in \{0,1\}^k$, and transforms it into a tamper-resilient circuit.

We describe our compiler in stages:

– First, we present a compiler that takes as input a circuit C_s and outputs a circuit $C_S^{(1)}$, which we partition into four segments. We prove that $C_S^{(1)}$ is secure against adversaries that tamper with at most a constant fraction of the memory gates, and tamper with at most a constant fraction of the wires in Segment 2, Segment 3, and Segment 4 of the circuit (and may tamper arbitrarily with Segment 1 of the circuit). We refer to such security as security against *local tampering*.

– Then, we show how to make the size of the memory, and the size of Segments 2-4, large enough so that the resulting circuit, denoted by $C_{\tilde{S}}^{(2)}$, is secure against adversaries who tamper with at most a constant fraction of the circuit *overall*. Namely, we prove that there is a constant $\lambda > 0$ such that the resulting circuit, $C_{\tilde{S}}^{(2)}$, is secure against any adversary that for each run of the circuit sends at most $\lambda \cdot |C_{\tilde{S}}^{(2)}|$ tampering instructions, where each tampering instruction is either a wire tampering or a memory tampering.

We note that both constructions have an AND gate (denoted by G_{cas}) which has a large fan-out. In Section 5 we show how using some additional ideas, one can bypass the need for such a gate.

4.1 The Compiler: First Construction

Let $\mathsf{ECC}(\cdot)$ be a binary error correcting code which can efficiently correct a constant fraction of errors $\delta > 0$. We denote the (efficient) decoding procedure by Dec.

In what follows we present a compiler that takes as input a circuit C_s and outputs a circuit $C_S^{(1)}$. The circuit $C_S^{(1)}$ takes as input a string $x \in \{0,1\}^n$ and outputs a bit b. In the case of no tampering, we show the correctness property: $C_S^{(1)}(x) = C_s(x)$. Moreover, we prove that the circuit $C_S^{(1)}$ is resilient to *local tampering*.

Remark. For simplicity we assume that $s = 0^k$ is "illegal." We note that this assumption is not necessary, and is made to simplify the construction and the analysis.

We next describe the circuit $C_S^{(1)}$, which we partition into four segments:

Memory: Encoding Secret s. The encoding $S = \mathsf{ECC}(s)$ is placed in the memory of $C^{(1)}$.

Segment 1: This segment consists of three sub-segments.

1. **Encoding Input x.** The first sub-computation encodes the public input x using ECC. Namely, it takes as input x and outputs $\mathsf{ECC}(x)$.
   We assume without loss of generality that $|\mathsf{ECC}(x)| = |\mathsf{ECC}(s)|$. This is without loss of generality since if, for example, $|x| < |s|$ then the encoding procedure will first artificially increase x by appending zeros to it, and then encode; and similarly if $|s| < |x|$.
2. **Circuit Computation.** The second sub-computation computes the output bit $b = C_s(x)$. Namely, it takes as input x and S and outputs $b = C_s(x)$, where $s = \mathsf{Dec}(S)$.
3. **PCPP Computation.** The third sub-computation computes a PCPP for the following pair language:

$$\mathcal{L} = \{(b, X \circ S) \mid \exists x \in \{0,1\}^n, s \in \{0,1\}^k \setminus \{0^k\} : X = ECC(x), S = ECC(s), b = C_s(x)\}$$

   Namely, it takes as input the pair $(b, X \circ S)$ and outputs π, which is a PCP of proximity for the statement $(b, X \circ S) \in \mathcal{L}$.
   The PCPP we use is one with randomness complexity $r = O(\log |C_s|)$, query complexity $q = O(1/\rho)$ for $\rho = \frac{\delta}{6}$,[5] perfect completeness, and soundness $1/2$, where the verifier rejects with probability at least $1/2$ statements $(b, X \circ S)$, for which $X \circ S$ is ρ-far from the language L_b. The existence of such a PCPP follows from Theorem 2 (see Section 2).

[5] Recall that δ is the fraction of errors that the error-correcting code ECC can correct.

We note that the outputs of each of the above sub-computations are used by many other subcomputations. Thus, each output wire of each of the above sub-computations is split into many output wires. These output wires all belong to Segment 2, except one output wire (which computes the bit $b = C_s(x)$, and belongs to Segment 4).

Segment 2: PCPP Verification. This segment consists of $\tau \triangleq 2^r = \mathrm{poly}(|C_s|)$ circuits, C_{v_i}, one for every possible PCPP verifier. Note that each verifier circuit has constant size, which we denote by $\tilde{\sigma}_{\mathsf{v}}$.[6] Each verifier circuit C_{v_i} takes as input q bits from $(X \circ S, b, \pi)$, and outputs a single bit β_i. All the output wires of the circuits C_{v_i} are fed as input to a single AND gate G_{cas} with fan-in τ. Thus, the output of G_{cas} is $\bigwedge_{1 \le i \le \tau} \beta_i$.
Let K be the number of bits in S, the encoding of the secret input s. The AND gate G_{cas} (from Segment 2) has fan-out $2K$, where the first K of the output wires belong to Segment 3 and the other K output wires belong to Segment 4. We denote the values of the output wires of G_{cas} by $\{\gamma_j\}_{j=1}^{2K}$.

Segment 3: Error Cascade. This segment contains only the first K output wires of G_{cas}, which have values $\{\gamma_j\}_{j=1}^{K}$. For every $j \in [K]$, the j'th output wire of G_{cas} is fed as input to memory gate j. Thus, if $\gamma_j = 0$, memory position j is overwritten with a 0. If $\gamma_j = 1$, memory position j is unchanged.

Segment 4: The Output. This segment consists of the rest of the circuit: The other K output wires of G_{cas}, which have values $\{\gamma_j\}_{j=K+1}^{2K}$, and a single AND gate G_{out} which has fan-in $K+1$ and outputs a single bit. G_{out} takes as input $\left(b, \{\gamma_j\}_{j=K+1}^{2K}\right)$ (all these input wires are part of this segment), and outputs $\tilde{b} = b \wedge \left(\bigwedge_{K+1 \le j \le 2K} \gamma_j\right)$.
We note that Segment 4 also includes the output wire of G_{out}, which is the final output of the circuit $C_S^{(1)}(x)$.

The compiled circuit $C_S^{(1)}$ is depicted in Figure 1.

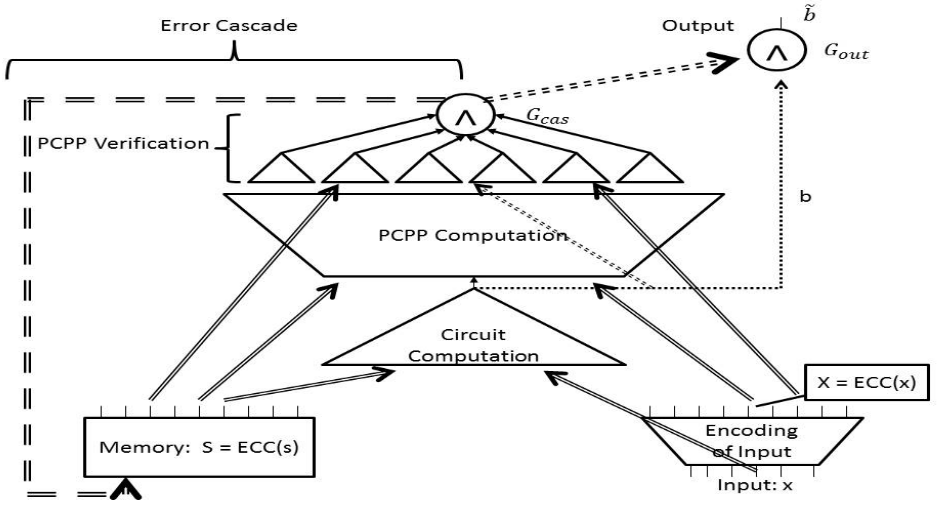

Fig. 1. The compiled circuit $C_S^{(1)}$. We show here the Memory and the 4 segments of the compiled circuit. Note that the encoding S and the input x are fed into the Circuit Computation Stage of Segment 1. Additionally, the encodings S and X are fed into the PCPP Computation Stage of Segment 1, and are fed to Segment 2 (the PCPP Verification Stage) along with the bit b, the output of the Circuit Computation Stage. The outputs of G_{cas} feed back into memory (this is Segment 3–Error Cascade) and to the final output gate G_{out} (in Segment 4) along with the output bit b of the Circuit Computation Stage. The final output of the circuit is $\tilde{b}$.

The circuit $C_S^{(1)}$ is secure against an adversary $\mathcal{A}$ who may tamper as follows:

- $\mathcal{A}$ can tamper with any number of wires in Segment 1 of the circuit (which consists of the encoding computation of the input x, the circuit computation $C_s(x)$, and the PCPP computation).

[6] We note that the amount of tampering we ultimately tolerate depends on $\tilde{\sigma}_{\mathsf{v}}$.

– $\mathcal{A}$ can tamper with at most λ-fraction of the wires in Segment 2 (PCPP verification), λ-fraction of the wires in Segment 3 (error cascade), λ-fraction of the wires in Segment 4 (output segment) and with at most λ-fraction of the memory gates, where $\lambda = \min\{\frac{1}{3\tilde{\sigma_V}}, \frac{\delta}{6}\}$.

We call such an adversary λ-*locally tampering*, and we denote the output of any such adversary by $\mathsf{LocalTAMP}_{\mathcal{A}}(C_S^{(1)})$.

Lemma 1. *Let* $\lambda = \min\{\frac{1}{3\tilde{\sigma_V}}, \frac{\delta}{6}\}$. *Then for any* λ-locally tampering *adversary* $\mathcal{A}$ *there exists a simulator* Sim, *that runs in time* $\mathrm{poly}(|\mathcal{A}|, |C_s|)$*, such that*

$$\{\mathsf{LocalTAMP}_{\mathcal{A}}(C_S^{(1)})\}_{k\in\mathbb{N}} \equiv \{\mathsf{LeakBB}_{\mathsf{Sim},\log T}(C_s)\}_{k\in\mathbb{N}},$$

where T *is an upper bound on the number of times that* $\mathcal{A}$ *runs the (tampered) circuit.*

In what follows, we give the high-level idea of the proof of Lemma 1, followed by the formal proof.

Proof idea. Fix any λ-*locally tampering* adversary $\mathcal{A}$. Let T be an upper bound on the number of times the adversary $\mathcal{A}$ runs a possibly tampered version of the circuit $C_s^{(1)}$. We construct a simulator Sim with black-box access to C_s, which is allowed to request $\log T$ bits of leakage on the secret s, and simulates the view of $\mathcal{A}$.

Sim chooses the leakage function to be the function $L : \{0,1\}^k \to \{0,1\}^{\log T}$, which has the adversary $\mathcal{A}$ hard-wired into it, and on input a secret s, outputs the largest index i^*, such that in the first $i^* - 1$ runs of the (possibly tampered) circuit by $\mathcal{A}$, all the (possibly tampered) input wires to the gate G_{cas} have the value 1.

After querying this leakage function and receiving an output i^*, Sim internally emulates $\mathcal{A}$, by emulating the output of each run of the circuit. Each time $\mathcal{A}$ calls the circuit with input x_i and a set of tampering instructions, the simulator Sim simulates the run of the (possibly tampered) circuit as follows:

– Simulate the output X_i' of the (possibly tampered) Encoding Input Segment. Efficiently decode X_i' to obtain $x_i' = \mathsf{Dec}(X_i')$.
  Note that X_i' may be far from a codeword, in which case x_i' may be $\perp$. Also, note that the simulator Sim is indeed able to carry out the above computation exactly as in the real computation, since this part of the computation does not use the secret s, and since Sim knows $\mathcal{A}$'s tampering instructions.
– If $i < i^*$, set all the input wires of G_{cas} to be 1, and set the output of the Circuit Computation Segment to be $b_i' = C_s(x_i')$. As we prove in Section 4.1, the fact that $i < i^*$ implies that it must be the case that $x_i' \neq \perp$, and that the output of the Circuit Computation Segment is indeed $C_s(x_i')$.
  Simulate the output b_{out} of G_{out}, using the tampering instructions of $\mathcal{A}$, assuming that the values of the $K+1$ incoming wires (before applying the tampering instructions to these wires) are $(1^K, b_i')$. Return b_{out}.
  Note that it is not necessarily the case that $b_{\mathsf{out}} = b_i'$ since the adversary $\mathcal{A}$ may tamper with some of the incoming wires of G_{out} and may tamper with the output wire.
– If $i \geq i^*$, return $b_{\mathsf{out}} = 0$, unless $\mathcal{A}$ tampers with the output wire, in which case Sim returns b if the tamper is "set to b", and returns 1 if the tamper is "toggle".
  We will argue that in this case, even if $\mathcal{A}$ tampers with the input wires to G_{out}, the output of G_{out} is always going to be 0, since he cannot tamper with all the input wires (recall that he can tamper with at most a λ-fraction of the wires). Thus, the output wire is always 0, unless the output wire itself is tampered with.

Proof of Lemma 1. Fix some λ-*locally tampering* adversary $\mathcal{A}$. Let T be an upper bound on the number of times the adversary $\mathcal{A}$ runs a possibly tampered version of the circuit $C_s^{(1)}$. We construct a simulator Sim with black-box access to C_s, which is allowed to request $\log T$ bits of leakage on the secret s, and simulates the view of $\mathcal{A}$.

Sim chooses the leakage function to be the function

$$L : \{0,1\}^k \to \{0,1\}^{\log T},$$

which has the adversary $\mathcal{A}$ hard-wired into it, and on input a secret s, outputs the largest index i^*, such that in the first $i^* - 1$ runs of the (possibly tampered) circuit by $\mathcal{A}$, all the (possibly tampered) input wires to the gate G_{cas} have the value 1.

After querying this leakage function and receiving an output i^*, Sim internally emulates $\mathcal{A}$, by emulating the output of each run of the circuit. Each time $\mathcal{A}$ calls the circuit with input x_i and a set of tampering instructions, the simulator Sim simulates the run of the (possibly tampered) circuit as follows:

– Simulate the output X'_i of the (possibly tampered) Encoding Input Segment. Efficiently decode X'_i to obtain $x'_i = \mathsf{Dec}(X'_i)$.
  Note that X'_i may be far from a codeword, in which case x'_i may be $\perp$. Also, note that the simulator Sim is indeed able to carry out the above computation exactly as in the real computation, since this part of the computation does not use the secret s, and since Sim knows $\mathcal{A}$'s tampering instructions.
– If $i < i^*$, set all the input wires of G_{cas} to be 1, and set the output of the Circuit Computation Segment to be $b'_i = C_s(x'_i)$. As we later claim, for every $i < i^*$ it must be the case that $x'_i \neq \perp$, and that the output of the Circuit Computation Segment is indeed $C_s(x'_i)$.
  Simulate the output b_{out} of G_{out}, using the tampering instructions of $\mathcal{A}$, assuming that the values of the $K+1$ incoming wires (before applying the tampering instructions to these wires) are $(1^K, b'_i)$. Return b_{out}.
  Note that it is not necessarily the case that $b_{\mathsf{out}} = b'_i$ since the adversary $\mathcal{A}$ may tamper with some of the incoming wires of G_{out} and may tamper with the output wire.
– If $i \geq i^*$, return $b_{\mathsf{out}} = 0$, unless $\mathcal{A}$ tampers with the output wire, in which case Sim returns b if the tamper is "set to b", and returns 1 if the tamper is "toggle".
  We will argue that in this case, even if $\mathcal{A}$ tampers with the input wires to G_{out}, the output of G_{out} is always going to be 0, since he cannot tamper with all the input wires (recall that he can tamper with at most a λ-fraction of the wires). Thus, the output wire is always 0, unless the output wire itself is tampered with.

Before analyzing the output distribution of Sim, we present the following claim:

Claim. For the i'th run of the (tampered) circuit by $\mathcal{A}$, let S'_i denote the memory contents, let X'_i denote the output of the Encoding Input Segment, and let b_i denote the output of the Circuit Computation Segment.[7] If all the input wires to G_{cas} have the value 1, then it must be the case that $(X'_i \circ S'_i)$ is ρ-close to some $(X''_i \circ S''_i)$ for which $(b_i, X''_i \circ S''_i) \in \mathcal{L}$.

Proof. Recall that $\mathcal{A}$ can tamper with at most a λ-fraction of the wires in Segment 2, and recall that the number of wires in Segment 2 is $\sigma_{\mathsf{v}} = \tilde{\sigma_{\mathsf{v}}} \cdot \tau$ (where $\tilde{\sigma_{\mathsf{v}}}$ is the number of wires in each PCPP verifier, and τ is the number of possible PCPP verifiers). Thus, $\mathcal{A}$ can tamper with at most $\lambda \cdot \tilde{\sigma_{\mathsf{v}}} \cdot \tau \leq \frac{\tau}{3}$ wires in Segment 2, and therefore can tamper with at most $1/3$ of the PCPP verifiers. This, together with the fact that all the input wires to G_{cas} are 1, implies that in run i, at least $2/3$ of the PCPP verifiers are not tampered with and output 1.

Thus, by the soundness of the PCPP (which is $1/2$), we must have that $(X'_i \circ S'_i)$ is ρ-close to some $(X''_i \circ S''_i)$ for which $(b_i, X''_i \circ S''_i) \in \mathcal{L}$, as desired.

We next argue that the random variable $\mathsf{LeakBB}_{\mathsf{Sim}, \log T}(C_s)$, which is the output of Sim, and the random variable $\mathsf{LocalTAMP}_{\mathcal{A}}(C_S^{(1)})$, which is the output of $\mathcal{A}$, are *identically* distributed. To this end, it is sufficient to argue that the output of the tampered circuit and the output of Sim are the same for all the runs of the circuit. There are two cases to consider:

Run i for $i < i^*$: In this case, Sim sets all the input wires of G_{cas} to 1, which must be correct since $i < i^*$. Sim also sets the output of the Circuit Computation segment to be $b'_i = C_s(x'_i)$, and simulates the output of the gate G_{out} exactly is in the real world execution of $\mathcal{A}$, assuming that the input wires to the gate G_{out} have the values $(1^K, b'_i)$, before applying the tampering instructions to these wires.
  Let b_i denote the output of the Circuit Computation Segment in the real experiment run by $\mathcal{A}$. The value outputted by Sim differs from the value outputted by the tampered circuit only if $b_i \neq C_s(x'_i)$. We next show that this cannot be the case.
  The fact that all the input wires of G_{cas} are 1 for $i < i^*$, together with Claim 4.1, implies that for all $i < i^*$, $(X'_i \circ S'_i)$ is ρ-close to some $(X''_i \circ S''_i)$ for which $(b_i, X''_i \circ S''_i) \in \mathcal{L}$.
  This implies, in particular, that for all $i < i^*$, X'_i and S'_i are 2ρ-close to the codewords X''_i and S''_i, respectively. This follows from the fact that X'_i and S'_i are of the same size. The fact that $2\rho < \delta$ implies that $x''_i = x'_i$, where x''_i is defined so that $\mathsf{ECC}(x''_i) = X''_i$ (and in particular that $x'_i \neq \perp$).
  We next claim that $S''_i = S = \mathsf{ECC}(s)$ for all $i < i^*$. We note that this claim, together with the fact that $(b_i, X''_i \circ S''_i) \in \mathcal{L}$, implies that indeed $b_i = C_s(x''_i) = C_s(x'_i) = b'_i$ for all $i < i^*$.

[7] We emphasize that S'_i, X'_i, b'_i are the values obtained *after* applying the tampering instructions of the i'th run.

Assume towards contradiction that there is some $i < i^*$ such that $S''_i = \mathsf{ECC}(\tilde{s})$ and $\tilde{s} \neq s$. Let $j < i^*$ be the *first* such i. Thus, S'_j and $\mathsf{ECC}(\tilde{s})$ are 2ρ-close. We first argue that it cannot be the case that $j = 1$. This follows from the fact that the adversary can tamper with at most λ-fraction of the memory, and thus S'_1 must be λ-close to $\mathsf{ECC}(s)$. If it was also 2ρ-close to $\mathsf{ECC}(\tilde{s})$, then it would imply that $\mathsf{ECC}(s)$ and $\mathsf{ECC}(\tilde{s})$ are $(2\rho + \lambda)$-close, contradicting the distance property of ECC, since $2\rho + \lambda < \frac{\delta}{3} + \frac{\delta}{6} < \delta$. Therefore it must be the case that $j > 1$. In particular, this implies that S'_{j-1} and $\mathsf{ECC}(s)$ are 2ρ-close.

The fact that $j < i^*$ implies that Error Cascade Segment (at the end of run $j - 1$) can overwrite at most λ-fraction of the memory. Moreover, the adversary in run j can tamper with at most λ-fraction of the memory gates. Thus, S'_{j-1} and S'_j are 2λ-close. We conclude that $\mathsf{ECC}(s)$ and $\mathsf{ECC}(\tilde{s})$ are $(4\rho + 2\lambda)$-close, contradicting the distance property of the ECC, since $4\rho + 2\lambda < \frac{4\delta}{6} + \frac{\delta}{3} = \delta$.

Run i for $i \geq i^*$: Suppose that one of the input wires of G_{cas} is 0 (and thus G_{cas} outputs 0^{2K} before applying the tampering instructions to these wires). If $\mathcal{A}$ does not tamper with the output wire, then it must be the case that the output of the tampered circuit is 0. This is the case since $\mathcal{A}$ can only tamper with λ-fraction of the wires in Segment 4, and thus cannot tamper with all the K input wires of G_{out} (which are all 0 before applying the tampering instructions to these wires). If $\mathcal{A}$ does tamper with the output wire, then Sim returns b if the tamper is "set to b", and returns 1 if the tamper is "toggle". Thus, Sim indeed perfectly simulates the output of the circuit in the case where one of the input wires of G_{cas} is 0. Therefore, it suffices to argue that for all $i > i^*$, one of the input wires of G_{cas} is 0, assuming this is the case for i^*.

Assume towards contradiction that for some $i > i^*$, all the input wires of G_{cas} are 1. Let $j > i^*$ be the smallest such index. Namely, assume that in the $j - 1$'st run one of the input wires of G_{cas} is 0, and yet in the j'th run all the input wires of G_{cas} are 1.

Again, using Claim 4.1, we have that in the j'th run $(X'_j \circ S'_j)$ is ρ-close to some $(X''_j \circ S''_j)$ for which $(b_j, X''_j \circ S''_j) \in \mathcal{L}$. This implies, in particular, that S'_j is at least 2ρ-close to some codeword.

However, since in run $j - 1$, one of the input wires of G_{cas} was 0, we have that at the end of round $j - 1$, the memory S'_{j-1} was updated so that at least $1 - \lambda$ fraction of the memory gates are overwritten with zeros. This is the case since $\mathcal{A}$ can tamper with at most λ-fraction of the wires in the Error Cascade Segment, where each such wire is used to update a memory gate. This, together with the fact that in run j the adversary $\mathcal{A}$ can tamper with at most λ-fraction of the memory, implies that S'_j is $(1 - 2\lambda)$-close to 0^K.

On the other hand, due to the distance property of ECC, any valid codeword S''_j can have at most a $(1-\delta)$-fraction of positions set to 0.[8] Thus, S'_j differs from any codeword in at least $(\delta - 2\lambda)$-fraction of positions, contradicting the fact that $\delta - 2\lambda \geq \delta - \frac{\delta}{3} = \frac{2\delta}{3} > 2\rho$.

□

4.2 The Compiler: Second Construction

Note that $C^{(1)}_S$ is secure against any adversary that tampers with only a constant fraction of the memory, and a constant fraction of the wires in Segments 2, 3 and 4. We would like to construct a circuit that is secure against adversaries that tamper with a constant fraction of the circuit overall. Namely, we would like our circuit, which will be denoted by $C^{(2)}_{\tilde{S}}$, to have the property that there is a constant $\lambda > 0$ such that $C^{(2)}_{\tilde{S}}$ is secure against any adversary that for each run of the circuit sends at most $\lambda \cdot |C^{(2)}_{\tilde{S}}|$ tampering instructions, where each tampering instruction is either a wire tampering or a memory tampering.

We do this by adding enough memory gates, and enough wires to the critical segments: the PCPP Verification Segment, the Error Cascade Segment and the Output Segment, so that the following two conditions hold.

1. Each of these segments, as well as the memory, remains resilient to a constant fraction of tampering.
2. The total number of wires in each of these segments, and the size of the memory, is a constant fraction of the total number of wires in the entire circuit.

More specifically, let $\sigma_{\mathsf{comp}} = \sigma_{\mathsf{comp}}(k)$ be the size of Segment 1 of the circuit $C^{(1)}_S$. We transform the circuit $C^{(1)}_S$, to ensure that the size $K = K(k)$ of the encoding of s in memory, the size $\sigma_{\mathsf{v}} = \sigma_{\mathsf{v}}(k)$ of Segment 2 (the PCPP

[8] Recall that we assume that $s = 0^k$ is not a valid seed, and thus $\mathsf{ECC}(0^k) = 0^K$ is not a valid codeword.

Verification Segment), the size $\sigma_{\mathsf{cas}} = \sigma_{\mathsf{cas}}(k)$ of Segment 3 (the Error Cascade Segment), and the size $\sigma_{\mathsf{out}} = \sigma_{\mathsf{out}}(k)$ of Segment 4 (the Output Segment), satisfy

$$\Theta(K(k)) = \Theta(\sigma_{\mathsf{v}}(k)) = \Theta(\sigma_{\mathsf{cas}}(k)) = \Theta(\sigma_{\mathsf{out}}(k)) = \Omega(\sigma_{\mathsf{comp}}(k)),$$

without changing the size σ_{comp} of Segment 1, and while keeping each of the above segments (Segments 2, 3, and 4), and the memory gates, resilient to constant fraction of tampering.

This transformation will allow us to prove security against adversaries who tamper with a constant fraction of the circuit overall. We proceed with the formal construction.

Let

$$M = M(k) = \max\{\sigma_{\mathsf{v}}(k), K(k), \sigma_{\mathsf{comp}}(k)\}.$$

Let $p_1(k) \triangleq M/\sigma_{\mathsf{v}}(k)$, and let $\sigma'_{\mathsf{v}}(k) \triangleq p_1(k) \cdot \sigma_{\mathsf{v}}(k)$. Similarly, let $p_2(k) \triangleq M(k)/K(k)$, and let $K'(k) \triangleq p_2(k) \cdot K(k)$. For the sake of simplicity we assume that p_1 and p_2 are integers.[9] Note that under this simplifying assumption $K' = \sigma'_{\mathsf{v}} = M$.

In what follows we give a formal description of our second compiler, which takes as input a (private) circuit C_s and outputs a (private) circuit $C^{(2)}_{\tilde{S}}$, defined as follows.

Memory: Encoding Secret s. Let $\widetilde{\mathsf{ECC}}(s) = \mathsf{ECC}(s)^{p_2(k)}$, where by $\mathsf{ECC}(s)^{p_2(k)}$ we denote $p_2(k)$ concatenated copies of $\mathsf{ECC}(s)$. Note that $\widetilde{\mathsf{ECC}}$ is an error-correcting code which has the same distance as ECC, which is at least 2δ. (The reason the distant of ECC is at least 2δ follows from the fact that it can correct up to δ-fraction of errors.) Let $\tilde{S} = \widetilde{\mathsf{ECC}}(s)$. We denote by $\tilde{S}(i,j)$ the j-th bit of the i-th copy of S. We denote by S the first row of the encoding $\tilde{S}$: $(\tilde{S}(1,j))_{j\in[K]}$. The encoding $\tilde{S}$ is placed in the memory of $C^{(2)}$.

Segment 1. This segment, which includes the encoding of the input x, the circuit computation, and the PCPP computation, remains exactly the same as Segment 1 of $C^{(1)}_S$. In order to keep this segment the same, instead of using the entire new memory $\widetilde{\mathsf{ECC}(S)}$, this segment only uses the first row of $\widetilde{\mathsf{ECC}(s)}$, which is exactly $S = \mathsf{ECC}(s)$, the memory content of $C^{(1)}$.

Segment 2: PCPP Verification. This stage consists of $\tau' \triangleq p_1(k)\cdot\tau$ number of circuits $\{C_{\mathsf{v}_{i,j}}\}_{i\in[\tau],j\in[p_1(k)]}$, where each PCPP verifier from $C^{(1)}_S$ is simply copied $p_1(k)$ times. Namely, for each $i \in [\tau]$ and each $j \in [p_1(k)]$, the circuit $C_{\mathsf{v}_{i,j}}$ is simply a copy of C_{v_i}. We note that each $C_{\mathsf{v}_{i,j}}$ only accesses the first row of memory $S = (\tilde{S}_{1,j})_{j\in[k]}$, and as before, each verifier circuit $C_{\mathsf{v}_{i,j}}$ takes as input a constant number of bits from $(X \circ S, b, \pi)$ and outputs a single bit $\beta_{i,j}$. Note that each verifier circuit still has constant size $\tilde{\sigma}_{\mathsf{v}}$.
All these τ' output wires of the circuits $C_{\mathsf{v}_{i,j}}$ are inputs to the AND gate G_{cas}. This gate has K' additional input wires that belong to Segment 3 below (where K' is the number of bits in $\tilde{S}$). The gate G_{cas} has $2K'$ output wires, and we denote the values on these wires by $\{\gamma_{\ell,m}\}_{i\in[p_2(k)],j\in[2K]}$. The first K' output wires (with values $\{\gamma_{\ell,m}\}_{i\in[p_2(k)],j\in[K]}$) belong to Segment 3, and the other K' output wires belong to Segment 4.

Segment 3: Error Cascade. This segment has two parts:

– A circuit $C_{\mathsf{code}_{i,j}}$ of constant size σ_{code} for each bit $\tilde{S}_{i,j}$ of $\tilde{S}$: $1 \le i \le p_2(k)$, $1 \le j \le K$:
Input: $\tilde{S}(1,j)$, $\tilde{S}(i,j)$.
Output: $\psi_{i,j} = \neg(\tilde{S}(1,j) \oplus \tilde{S}(i,j))$.
All these output wires with values $\psi_{i,j}$ are inputs to G_{cas}. Thus, in total, G_{cas} has $K' + \tau'$ input wires (τ' from Segment 2 and K' from Segment 3), and it outputs

$$\left(\bigwedge_{i\in[\tau],j\in[p_1(k)]} \beta_{i,j}\right) \wedge \left(\bigwedge_{i\in[p_s(k)],j\in[K]} \psi_{i,j}\right)$$

[9] This simplifying assumption makes the analysis somewhat cleaner. Without this assumption, we would need to define $p_1(k) \triangleq \lceil M/\sigma_{\mathsf{v}}(k)\rceil$ and $p_2(k) \triangleq \lceil M(k)/K(k)\rceil$.

– The first K' output wires of G_{cas}, denoted by $(\gamma_{\ell,m})_{\ell\in[p_2(k)],m\in[K]}$, are fed to the memory gates. More specifically, for every $\ell \in [p_2(k)]$ and $m \in [K]$, the (ℓ, m)-th output $\gamma_{\ell,m}$ of G_{cas} is the input to memory gate (ℓ, m). Thus, if $\gamma_{\ell,m} = 0$, memory position (ℓ, m) is overwritten with a 0. If $\gamma_{\ell,m} = 1$, memory position (ℓ, m) is unchanged.

Segment 4: The Output. This segment of the circuit consists of K' output wires of G_{cas}, which have the values $\{\gamma_{\ell,m}\}_{\ell\in[p_2(k)],m\in[K+1,2K]}$. This segment has a single AND gate G_{out} which has fan-in $K' + 1$. This segment contains all the $K' + 1$ input wires to G_{out}: The first K' input wires are exactly the K' output wires of G_{cas} (with values $\{\gamma_{\ell,m}\}_{\ell\in[p_2(k)],m\in[K+1,2K]}$), and the other input wire of G_{out} is an output wire of the Circuit Computation in Segment 1, which computes the bit $b = C_s(x)$. The AND gate G_{out} has a single output wire which is

$$\tilde{b} = b \wedge \left(\bigwedge_{\ell\in[p_2(k)],m\in[K'+1,2K']} \gamma_{\ell,m} \right).$$

The final output of the circuit $C^{(2)}_{\tilde{S}}(x)$ is $\tilde{b}$.

The compiled circuit $C^{(2)}_{\tilde{S}}$ is depicted in Figure 2.

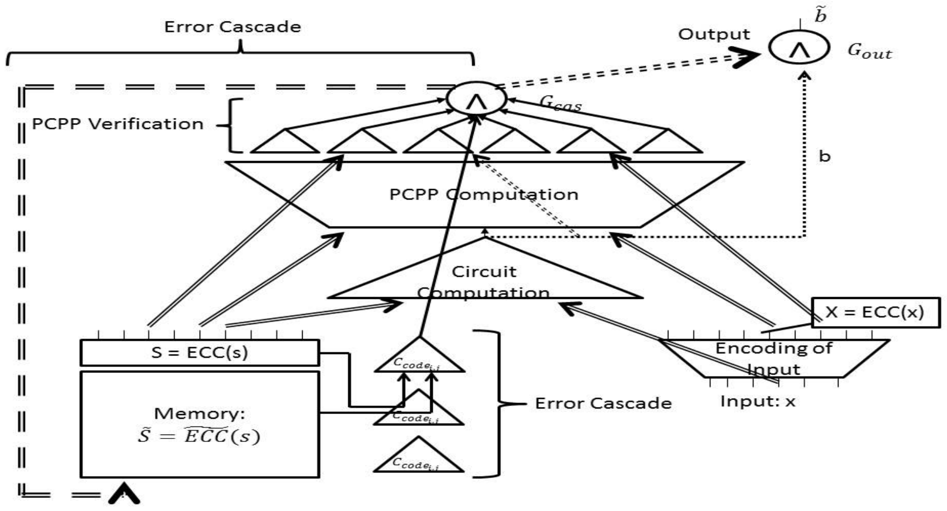

Fig. 2. The compiled circuit $C^{(2)}_{\tilde{S}}$. We show here the Memory and 4 segments of the compiled circuit. Note the differences between $C^{(2)}_{\tilde{S}}$ and $C^{(1)}_{S}$: First, the encoding $\tilde{S} = \widetilde{\mathsf{ECC}}(s)$ is placed in memory and the size of memory is increased. Second, the size of Segment 2 (PCPP Verification), Segment 3 (Error Cascade), and Segment 4 (Output), is increased. Third, the constant-size circuits $C_{\mathsf{code}_{i,j}}$ are added to check consistency of the encoding. The outputs of these circuits are then all inputted to G_{cas}.

Recall that we denote by $\sigma'_{\mathsf{v}} = \sigma'_{\mathsf{v}}(k)$ the size of the PCPP Verification Segment in $C^{(2)}_{\tilde{S}}$, and we denote by K' the number of memory gates in $C^{(2)}_{\tilde{S}}$. We also denote by $\sigma'_{\mathsf{cas}} = \sigma'_{\mathsf{cas}}(k)$ the size of the Error Cascade Segment in $C^{(2)}_{\tilde{S}}$ and denote by $\sigma'_{\mathsf{out}} = \sigma'_{\mathsf{out}}(k)$ be the size of the Output Segment in $C^{(2)}_{\tilde{S}}$. It follows by construction that

$$\Theta(\sigma'_{\mathsf{v}}(k)) = \Theta(\sigma'_{\mathsf{cas}}(k)) = \Theta(\sigma'_{\mathsf{out}}(k)) = \Theta(K'(k)) = \Omega(\sigma_{\mathsf{comp}}(k)),$$

as desired.

In what follows, we denote by σ' the total size of the circuit $C^{(2)}_{\tilde{S}}$.

Let

$$\alpha = \alpha(k) = \min\left\{ \frac{\sigma'_{\mathsf{v}}(k)}{\sigma'(k)}, \frac{\sigma'_{\mathsf{cas}}(k)}{\sigma'(k)}, \frac{\sigma'_{\mathsf{out}}(k)}{\sigma'(k)} \right\}.$$

Note that $\alpha(k) = \Theta(1)$.

Theorem 3. *Let $\lambda' = \min\{\alpha\frac{1}{3\tilde{\sigma_{\mathsf{v}}}}, \alpha\frac{\delta}{6\sigma_{\mathsf{code}}}\}$. Then $C^{(2)}_{\tilde{S}}$ is secure against λ'-tampering adversaries (as defined in Definition 5).*

In what follows we prove Theorem 3. We note that in the proof we use that fact that every λ'-*globally tampering* adversary in $C^{(2)}_{\tilde{S}}$ is also a $\lambda = \min\{\frac{1}{3\tilde{\sigma_{\mathsf{v}}}}, \frac{\delta}{6\sigma_{\mathsf{code}}}\}$-*locally tampering* adversary in $C^{(2)}_{\tilde{S}}$.

Proof. We prove the stronger statement that, in fact for $\lambda = \min\{\frac{1}{3\tilde{\sigma_{\mathsf{v}}}}, \frac{\delta}{6\sigma_{\mathsf{code}}}\}$, it holds that for every λ-*locally tampering* adversary $\mathcal{A}$ there exists a simulator Sim such that

$$\{\mathsf{TAMP}_{\mathcal{A}}(C^{(2)}_{\tilde{S}})\}_{k\in\mathbb{N}} \equiv \{\mathsf{LeakBB}_{\mathsf{Sim},\log T}(C_s)\}_{k\in\mathbb{N}},$$

where T is an upper bound on the number of times that $\mathcal{A}$ runs the tampered circuit. This suffices to obtain our theorem since any λ'-tampering adversary is in particular λ-locally tampering.

We fix any λ-locally tampering adversary $\mathcal{A}$, and we construct a simulator Sim with black-box access to C_s, which is allowed to request $\log T$ bits of leakage.

The leakage function L chosen by Sim is the same function as in the proof of Lemma 1, but with respect to the circuit $C^{(2)}_{\tilde{S}}$ instead of $C^{(1)}_{S}$. Namely, the leakage function

$$L : \{0,1\}^k \to \{0,1\}^{\log T},$$

has the adversary $\mathcal{A}$ hard-wired into it, and on input a secret s, outputs the largest index i^*, such that in the first i^*-1 runs of the (possibly tampered) circuit by $\mathcal{A}$, all the (possibly tampered) input wires to the gate G_{cas} have the value 1.

The simulator simulates $\mathcal{A}$ exactly as is done in the proof of Lemma 1. Namely, each time $\mathcal{A}$ calls the circuit with input x_i and a set of tampering instructions, the simulator simulates the output as follows: It simulates the output X'_i of the (possibly tampered) Encoding Input Segment, and computes the decoding $x'_i = \mathsf{Dec}(X'_i)$.[10]

If $i < i^*$, the simulator Sim sets the value of all the input wires of G_{cas} to be 1, and sets the output of the Circuit Computation Segment to be $b'_i = C_s(x'_i)$. Then, it simulates the output b_{out} of G_{out}, assuming that the $K'+1$ incoming wires (before applying the tampering instructions to these wires) have values $(1^{K'}, b'_i)$.

If $i \geq i^*$ then Sim returns $b_{\mathsf{out}} = 0$, unless $\mathcal{A}$ tampers with the output wire, in which case Sim returns b if the tamper is "set to b", and returns 1 if the tamper is "toggle".

Before analyzing the output distribution of Sim, we present the following claim:

Claim. For the i'th run of the (tampered) circuit by $\mathcal{A}$, let $\tilde{S}'_i$ denote the contents of memory, let X'_i denote the output of the Encoding Input Segment, and let b_i denote the output of the Circuit Computation Segment.[11] If all the input wires to G_{cas} have the value 1, then the following two conditions hold:

(1) $(X'_i \circ S'_i)$ is ρ-close to some $(X''_i \circ S''_i)$ for which $(b_i, X''_i \circ S''_i) \in \mathcal{L}$, where $S'_i \triangleq (\tilde{S}'_i(1,j))_{j\in[K]}$ is the first row of $\tilde{S}'_i$.

(2) Suppose $S''_i = \mathsf{ECC}(s^*)$.[12] Then $\tilde{S}'_i$ is $(2\rho + \delta/3)$-close to the codeword $\tilde{S}''_i = \widetilde{\mathsf{ECC}}(s^*)$.

[10] Note that X'_i may be far from a codeword, in which case x'_i may be $\perp$.

[11] We emphasize that $\tilde{S}'_i, X'_i, b'_i$ are the values obtained *after* applying the tampering instructions of the i'th run.

[12] Note that the fact that $(b_i, X''_i \circ S''_i) \in \mathcal{L}$ implies that S''_i is indeed a codeword, and hence there must exist s^* such that $S''_i = \mathsf{ECC}(s^*)$.

Proof. We first prove Condition (1), whose proof is essentially the same as the proof of Claim 4.1. Assume towards contradiction that $(X'_i \circ S'_i)$ is ρ-far from every $(X''_i \circ S''_i)$ for which $(b_i, X''_i \circ S''_i) \in \mathcal{L}$. Recall that $\mathcal{A}$ can tamper with at most λ-fraction of the wires in Segment 2. Note that now the size of Segment 2 is $\sigma'_{\mathsf{v}} = \tilde{\sigma_{\mathsf{v}}} \cdot \tau \cdot p_1$ (where $\tilde{\sigma_{\mathsf{v}}}$ is the size of each PCPP verifier, and $\tau \cdot p_1$ is the number of PCPP verifiers). Thus, $\mathcal{A}$ can tamper with at most $\lambda \cdot \tilde{\sigma_{\mathsf{v}}} \cdot \tau \cdot p_1 \leq \frac{p_1 \cdot \tau}{3}$ wires in Segment 2. This, together with the fact that all the input wires to G_{cas} are 1, implies that in run i, at least $2/3$ of the PCPP verifiers are not tampered with and output 1.

Thus, by the soundness of the PCPP (which is $1/2$), we must have that $(X'_i \circ S'_i)$ is ρ-close to some $(X''_i \circ S''_i)$ for which $(b_i, X''_i \circ S''_i) \in \mathcal{L}$, as desired.

We next prove Condition (2). We start by noting that Condition (1) implies that S'_i is 2ρ-close to some codeword $S''_i = \mathsf{ECC}(s^*)$. This follows from the assumption that $|X'_i| = |S'_i|$. Assume for the sake of contradiction that $\tilde{S}'_i$ is not $(2\rho + \delta/3)$-close to $\tilde{S}''_i = \widetilde{\mathsf{ECC}}(s^*)$. Then it must be the case that $(S'_i)^{p_2(k)}$ and $\tilde{S}'_i$ are not $\delta/3$-close, since $(S'_i)^{p_2(k)}$ is 2ρ-close to $\widetilde{\mathsf{ECC}}(s^*)$. Consider the K' wires $\psi_{i,j}$ which are all inputted to the AND gate G_{cas}. It must be the case that $\delta/3$-fraction of these wires have the value 0 (before applying the tampering instructions to these wires). Thus, $\delta K'/3$ of these input wires to G_{cas} have value 0 (before applying the tampering instructions to these wires). However, $\mathcal{A}$ can tamper with at most $\lambda \leq \frac{\delta}{6\sigma_{\mathsf{code}}}$-fraction of the total number of wires in Segment 3, where the total number of wires in Segment 3 is

$$\sigma'_{\mathsf{cas}} = \sigma_{\mathsf{code}} \cdot K' + K'.$$

Thus, $\mathcal{A}$ can tamper with at most

$$\delta K'/6 + (\delta K')/(6\sigma_{\mathsf{code}}) < \delta K'/6 + \delta K'/6 = \delta K'/3$$

of the wires in Segment 3. So even after tampering, at least one of the input wires to G_{cas} will be 0, contradicting the assumption that for every $i < i^*$ all the (possibly tampered) input wires to G_{cas} are 1.

To argue that the output of the simulator Sim is identically distributed to the output of $\mathcal{A}$, we distinguish between two cases:

Run i for $i < i^*$: In this case, to prove correctness of the simulation we need to prove that in the real run of $\mathcal{A}$, it holds that $x'_i \neq \perp$ and that the output of the Circuit Computation Segment is $b_i = C_s(x'_i)$.

Recall that for all $i < i^*$, all the (possibly tampered) input wires of G_{cas} have the value 1. This, together with Condition (1) of Claim 4.2, implies that for all $i < i^*$, X'_i and S'_i are 2ρ-close to codewords X''_i and S''_i, respectively. The fact that $2\rho < \delta$ implies that $x''_i = x'_i$, where x''_i is defined so that $\mathsf{ECC}(x''_i) = X''_i$ (and in particular that $x'_i \neq \perp$).

We next claim that $\tilde{S}''_i = \tilde{S} = \widetilde{\mathsf{ECC}}(s)$ for all $i < i^*$. We note that this claim, together with Condition (1) of Claim 4.2 (which guarantees that $(b_i, X''_i \circ S''_i) \in \mathcal{L}$ and that $x''_i = x'_i$), implies that indeed $b_i = C_s(x''_i) = C_s(x'_i)$ for all $i < i^*$, as desired.

Condition (2) of Claim 4.2 implies that for every $i < i^*$, $\tilde{S}'_i$ is $(2\rho + \delta/3)$-close to some codeword $\tilde{S}''_i = \widetilde{\mathsf{ECC}}(s')$. Assume that for some $i < i^*$, $s' \neq s$, and let $j < i^*$ be the *first* such i. We first note that it must be the case that $j > 1$. This follows from the fact that $\tilde{S}'_1$ is λ-close to $\tilde{S} = \widetilde{\mathsf{ECC}}(s)$. Moreover, the distance between any two codewords in $\widetilde{\mathsf{ECC}}$ is at least δ, and thus $\tilde{S}'_1$ is $(\delta - \lambda)$-far from any other codeword, where $\delta - \lambda > \frac{5\delta}{6}$ is greater than $2\rho + \delta/3 = \frac{2\delta}{6} + \frac{\delta}{3} = \frac{2\delta}{3}$.

Thus, $j > 1$. Namely, on the one hand, $\tilde{S}'_j$ and $\widetilde{\mathsf{ECC}}(s')$ are $(2\rho + \delta/3)$-close; and on the other hand, $\tilde{S}'_{j-1}$ and $\widetilde{\mathsf{ECC}}(s)$ are $(2\rho + \delta/3)$-close. This together with the fact that every two codewords are 2δ-far (follows from the fact that $\widetilde{\mathsf{ECC}}$ can correct up to δ errors), implies that $\tilde{S}'_{j-1}$ and $\tilde{S}'_j$ are $(2\delta - 4\rho - \frac{2\delta}{3})$-far, where $2\delta - 4\rho - \frac{2\delta}{3} = 2\delta - \frac{2\delta}{3} - \frac{2\delta}{3} = \frac{2\delta}{3}$.

We reach a contradiction by arguing that $\tilde{S}'_{j-1}$ and $\tilde{S}'_j$ are $\frac{\delta}{2}$-close. At the end of the $j-1$'st run of the (possibly tampered) circuit, the memory $\tilde{S}'_{j-1}$ may be updated. We argue that at most $\lambda(K' + K'\sigma_{\mathsf{code}})$ of the memory gates are overwritten. This is the case since $j - 1 < i^*$, and thus all the K' wires updating the memory (which are output wires of G_{cas}) are 1 before applying the tampering instructions to these wires. Note that these K' wires belong to Segment 3 of the circuit, a segment which consists of $K' + K'\sigma_{\mathsf{code}}$ wires. Since the adversary is λ-locally tampering, $\mathcal{A}$ can tamper with at most λ-fraction of Segment 3, and thus can tamper with at most

$\lambda(K' + K'\sigma_{\mathsf{code}})$ of the K' wires that update the memory. Moreover, note that $\mathcal{A}$ can tamper with the values of at most a λ-fraction of positions in memory during the j'th run. Thus, all in all, $\tilde{S}'_{j-1}$ and $\tilde{S}'_j$ differ in at most $\lambda(K' + K'\sigma_{\mathsf{code}}) + \lambda K'$ memory locations, where

$$\lambda(K' + K'\sigma_{\mathsf{code}}) + \lambda K' = \lambda K'\sigma_{\mathsf{code}} + 2\lambda K' \leq \frac{\delta K'}{6} + \frac{\delta K'}{3} = \frac{\delta K'}{2} \tag{4.1}$$

(follows from the fact that $\lambda \leq \frac{\delta}{6\sigma_{\mathsf{code}}}$).

Thus, we conclude that $\tilde{S}'_{j-1}$ and $\tilde{S}'_j$ are $\frac{\delta}{2}$-close, in contradiction to the above.

Run i for $i \geq i^*$: As was argued in the proof of Lemma 1, if one of the (possibly tampered) input wires of G_{cas} has the value 0 then the simulation is correct. Thus, it suffices to argue that indeed one of the (possibly tampered) input wires of G_{cas} has the value 0 in each run $i \geq i^*$. Assume towards contradiction that there exists $j > i^*$ such that in the j'th run all the (possibly tampered) input wires of G_{cas} have the value 1, and yet in the $j-1$'st run one of the (possibly tampered) input wires of G_{cas} has the value 0.

Condition (2) of Claim 4.2 states that if in run j, all the (possibly tampered) input wires of G_{cas} have the value 1, then $\tilde{S}'_j$ is $(2\rho+\delta/3)$-close to some codeword $\tilde{S}''_j = \widetilde{\mathsf{ECC}}(s'')$. Moreover, $\widetilde{\mathsf{ECC}}$ had distance 2δ, and thus any valid codeword $\tilde{S}''_j$ can have at most a $(1-2\delta)$-fraction of positions set to 0.[13] This implies that $\tilde{S}'_j$ can have at most $(1 - 2\delta + 2\rho + \delta/3)$-fraction of positions set to 0, where

$$1 - 2\delta + 2\rho + \delta/3 = 1 - 2\delta + \frac{2\delta}{3} = 1 - \frac{4\delta}{3}.$$

Thus, $\tilde{S}'_j$ is $\frac{4\delta}{3}$-far from $0^{K'}$.

On the other hand, the fact that in run $j-1$, one of the (possibly tampered) input wires of G_{cas} has the value 0, implies that at the end of round $j-1$, at least $\lambda(K'+K'\sigma_{\mathsf{code}})$ of the memory gates are overwritten and set to 0. This is the case since the K' wires updating the memory all have the value 0 before applying the tampering instructions to these wires. These wires belong to Segment 3 of the circuit, a segment which consists of $K' + K'\sigma_{\mathsf{code}}$ wires. Thus, the adversary can tamper with at most $\lambda(K' + K'\sigma_{\mathsf{code}})$ of these wires. In addition the adversary can tamper with at most λ-fraction of the memory gates in the j'th run, and thus $\tilde{S}'_j$ can have at most $\lambda(K'+K'\sigma_{\mathsf{code}})+\lambda K' \leq \frac{\delta K'}{2}$ non-zero memory gates (see Equation (4.1)). Hence, $\tilde{S}'_j$ is $\frac{\delta}{2}$-close to $0^{K'}$, contradicting the fact that it is $\frac{4\delta}{3}$-far from $0^{K'}$.

5 Dealing with Large Fan-out

In practice, circuit gates do not have large fan-out. Rather, gates have a single output wire which is then inputted into a device called a *splitter* that replicates the wire many times. We now consider a model where all gates have fan-out 1 (possibly feeding into splitters) and where each input and output wire of a splitter may be individually tampered with.

It is not hard to see that we can safely replace all gates with large fan-out in our construction with splitters, *except* for the gate G_{cas}. Indeed, if we modify the high fan-out gate G_{cas} to have a a single output wire and a splitter then the adversary can simply set the value of the single output wire of G_{cas} to 1 in each run of the circuit, and thus prevent self-destruct from occuring, which means that security cannot possibly hold.

In order to ensure that each gate has a single output wire (which is possibly fed into a splitter), we eliminate the G_{cas} gate altogether, and modify the rest of the circuit. Intuitively, we add an additional matrix M to the memory, which is initialized to 1 and it "keeps track" of whether self-destruct has previously occurred. When self-destruct occurs M is overwritten by 0's. Moreover, we will show how to construct a circuit on top of M in such a way that once M is set to have a large constant fraction of 0's in some run ℓ, M is guaranteed to have a large constant fraction of 0 for *every* run $i > \ell$.

More specifically, we add to the memory a matrix M of size $p_2(k) \times K = K'$ with entries all initialized to 1. We use an expander graph to update the memory M.

[13] Recall that we assume that $s = 0^k$ is not a valid seed, and thus $\widetilde{\mathsf{ECC}}(0^k) = 0^{K'}$ is not a valid codeword.

Consider a bipartite expander graph $\mathcal{G}_M$ of constant degree d on vertex set $(M^{\mathsf{after}}, M^{\mathsf{before}})$ where $|M^{\mathsf{after}}| = |M^{\mathsf{before}}| = K'$. For each i, j, the vertices $M^{\mathsf{before}}_{i,j}, M^{\mathsf{after}}_{i,j}$ are identified with memory location $M_{i,j}$. Intuitively, the vertex set M^{before} corresponds to the memory locations and values before some run i and M^{after} corresponds to the memory locations and values after run i has completed. For every edge between two vertices $M^{\mathsf{after}}_{i,j}$ and $M^{\mathsf{before}}_{k,\ell}$ in $\mathcal{G}_M$ we connect the output of $M_{k,\ell}$ to the input of $M_{i,j}$. Ultimately, each memory gate $M_{i,j}$ will have a constant number (d) of input wires and will be "zeroed out" iff at least one of its input wires is set to 0. Note that since degree d is a constant, total number of wires added to our previous construction is $d \cdot K' = \Theta(K')$.

A high-level overview of the construction is presented below. We present the Segments of the circuit in the order in which they are computed. Namely, the output of Segment 4 is computed before Segments 2 and 3. Details follow.

Memory: Encoding of s and 1 string. In addition to $\tilde{S} = \widetilde{\mathsf{ECC}}(s)$ in memory, where $|\tilde{S}| = K'$, we add to memory a matrix M of size $p_2(k) \times K = K'$ with entries all initialized to 1.
We denote the (i, j)'th position in $\tilde{S}$ by $\tilde{S}_{i,j}$ and the (i, j)'th position in M by $M_{i,j}$.

Segment 1. This segment, which includes the encoding of the input x, the circuit computation, and the PCPP computation, remains the same as Segment 1 of $C^{(2)}_{\tilde{S}}$.

Segment 4: The Output. The AND gate G_{out} has $K' + 1$ input wires. The first K' input wires are the output wires from each memory gate $M(i, j)$ with value $m_{i,j}$. The last input wire is the output wire from the circuit computation in Segment 1 with value b. The gate G_{out} outputs $2K' + 1$ wires (i.e. a single output wire and a splitter into $2K' + 1$ wires) with value:

$$\tilde{b} = b \wedge \left(\bigwedge_{i \in [p_2(k)], j \in [K]} m_{i,j} \right).$$

$2K'$ of the output wires are inputted to their respective PCPP Verifiers. The remaining wire is the final output value of the circuit.

Segment 2: PCPP Verification. We increase the number of verifiers so there are exactly K' number of verifiers. Otherwise, this segment remains exactly the same as Segment 2 of $C^{(2)}_{\tilde{S}}$, with the following exceptions:

– Each PCPP Verifier circuit $C_{\mathsf{v}_{i,j}}$ takes the input bit $\tilde{b}$ from the output of gate G_{out}, instead of taking the bit b, from the Circuit Computation Stage in Segment 1.
– The output wires of the PCPP Verifier circuits $C_{\mathsf{v}_{i,j}}$ with values $\{\gamma_{i,j}\}_{i \in [p_2[k]], j \in [K]}$ are not inputted to the gate G_{cas}, but rather directly to memory gates M (see Segment 3 below).

Segment 3: Error Cascade. This segment has two parts as before:

– The first part is the same as in the previous construction, except that all the output wires of $C_{\mathsf{code}_{i,j}}$ with values $\psi_{i,j}$ are fed into the second part of memory. More specifically, the output wire of $C_{\mathsf{code}_{i,j}}$ is inputted to memory location $M_{i,j}$.
– The K' output wires of $C_{\mathsf{v}_{i,j}}$, with values $(\gamma_{i,j})$ are fed to the memory gates. More specifically, for every $i \in [p_2(k)]$ and $j \in [K]$, the (i, j)'th output $\gamma_{i,j}$ is the input to memory gate $\tilde{S}_{i,j}$ and for every $i \in [p_2(k)]$ and $j \in [K]$, the (i, j)'th output $\gamma_{i,j}$ is the input to memory gate $\tilde{S}_{i,j}$.
– Finally, we specify the wiring on the memory gates M induced by the edges of the expander graph $\mathcal{G}_M$. For every pair of memory gates $M_{i,j}$, $M_{k,\ell}$, there is a wire going from the output of $M_{k,\ell}$ to the input of $M_{i,j}$ iff there is an edge between $M^{\mathsf{after}}_{i,j}$ and $M^{\mathsf{before}}_{k,\ell}$ in $\mathcal{G}_M$

We now give some intuition for why this construction is secure. The leakage function used by the simulator is now slightly different. The simulator asks for the largest index i^*, such that in all runs $i < i^*$, the value of the (possibly tampered) single output wire of the gate G_{out} is the "correct" value; i.e., it is $C_s(x')$, where x' is the input extracted by the simulator, as described in the simulation in the proof of Theorem 3 and s is the original secret.

Note that simulating runs $i \leq i^*$ is easy, since knowing the output of G_{out}, the simulator can perfectly simulate the final output of the circuit, by observing the tampering instructions that are made to the final output wire of the circuit.

For the simulation of runs $i > i^*$, we prove that for every run $i > i^*$, it must be the case that at least one (possibly tampered) input wire to G_{out} has the value 0. Once we establish this, again simulating becomes easy since the simulator can perfectly simulate the final output of the circuit by observing the tampering instructions to the output wires of G_{out}.

Thus, it remains to argue that indeed for every run i where $i > i^*$ at least one (possibly tampered) input wire to G_{out} has value 0.

We first argue that there must be some first index $\ell \le i^*$ such that at the end of run ℓ, at least a $1-\eta$-fraction of M is set to 0[14].

Since the leakage function returns i^* such that $b_{i^*} \neq C_s(x'_{i^*})$, there are two cases to consider:

Case 1: $(X'_{i^*} \circ S'_{i^*})$ is ρ-far from every $(X''_{i^*} \circ S''_{i^*})$ for which $(b_{i^*}, X''_{i^*} \circ S''_{i^*}) \in \mathcal{L}$. In this case, we have that at least a $1-\eta/2$-fraction of PCPP verifiers output 0 in run i^*. Thus, since the tampering rate is $<< \eta/4$, we have that at the end of run i^* at least a $1-\eta$-fraction of M is set to 0.

Case 2: $(X'_{i^*} \circ S'_{i^*})$ is ρ-close to some $(X''_{i^*} \circ S''_{i^*})$ for which $(b_{i^*}, X''_{i^*} \circ S''_{i^*}) \in \mathcal{L}$, but $S''_{i^*} \neq \mathsf{ECC}(s)$. In this case, we argue that there must be some first index $\ell \le i^*$ such that at the end of the ℓ'th run at least a $1-\eta$-fraction of M is set to 0.

Again, we must consider two cases: Either for some $\ell < i^*$ we have that S'_ℓ is ρ-far from every codeword $S'' = \mathsf{ECC}(s'')$, or for every $\ell \le i^*$ we have that S'_ℓ is ρ-close to some codeword $S'' = \mathsf{ECC}(s'')$.

In the former case, we have (as in Case 1) that at least a $1-\eta/2$-fraction of PCPP verifiers output 0 in run ℓ. Thus, since the local tampering rate is $<< \eta/4$, we have that at the end of run ℓ at least a $1-\eta$-fraction of M is set to 0.

In the latter case, let $\ell \le i^*$ be the *first* index such that $S'_{\ell-1}$ is ρ-close to $S = \mathsf{ECC}(s)$ and S'_ℓ is ρ-close to $S'' = \mathsf{ECC}(s'') \neq S$. Let $\mu_{\ell-1}$ denote the fraction of circuits $C^{i,j}_{\mathsf{code}}$ which output 0 after run $\ell-1$ and let μ_ℓ denote the fraction of circuits $C^{i,j}_{\mathsf{code}}$ which outputted 1 after run $\ell-1$ and output 0 after run ℓ. We claim that $\mu + \mu_{m+1} \ge \delta - 2\rho - 2\lambda$, where δ is the distance of the code $\widetilde{\mathsf{ECC}}$, and λ is the local tampering rate. Moreover, since only λ memory gates M can be tampered in each of these two runs, we must have that at least a $(\delta - 2\rho - 4\lambda) \ge (1-\eta)$-fraction of memory gates M are indeed set to 0 after the ℓ'th run.

Now, as above, let $\ell \le i^*$ denote the first index such that $(1-\eta)$-fraction of the memory cells M are set to 0 after the ℓ'th run. We claim that for every run $j > \ell$ it is the case that at the end of run j, M has at least $(1-\eta)$-fraction of 0's, which in turn implies that there is at least one input wire to G_{out} set to 0 in each run $j > \ell$. We argue this by leveraging the properties of the bipartite expander graph $\mathcal{G}_M$.

Let $\mathcal{G}_M$ have expansion factor of ξ on all sets $L \subseteq M^{\mathsf{after}}$ where $|L| \le \eta \cdot K'$ and $\xi > 0, \eta > 0$ are constants. Assume towards contradiction that right before run j, M has at most η-fraction of 1's and that right before run $j+1$, M has at least η-fraction of 1's, for some $j > \ell$.

Let $L \subseteq M^{\mathsf{after}}$ denote a set of memory locations in M that are set to 1 right before run $j+1$, such that $|L| = \eta \cdot K'$. Let $R \subseteq M^{\mathsf{before}}$ denote the neighborhood of R. Note that by the expansion property, we have that $|R| \ge \xi \cdot |L|$. Thus, there must be at least $|R| - |L| = \xi \cdot \eta \cdot K'$ number of positions in R that were set to 0 right before the j'th run (since there were a total of at most $\eta \cdot K'$ positions total set to 1 before the j'th run). Thus, without tampering, at least $\xi \cdot \eta/d \cdot K'$ number of positions in L would have been set to 0. So in order to ensure that these positions in L are set to 1, the adversary must tamper with at least $\xi \cdot \eta/d \cdot K'$ number of wires or memory locations. However, this is impossible, since we choose the tampering rate λ to be a constant small enough that the adversary cannot tamper with a $\xi \cdot \eta/d$-fraction of the K' locations in a single run.

6 Security against Leakage and Tampering

In this section, we give a high-level overview of how to convert any circuit C_s to a functionally equivalent circuit, which is secure against both tampering (as defined in Section 3) and *leakage*. We warn the reader that this section does not contain any formal theorems or proofs, and contains only high-level ideas. The reason we do not formalize these ideas is that the results we obtain are quite weak: The resulting circuit is only resilient to tampering with $1/\operatorname{poly}(k)$ of the wires, where k is the security parameter. However, we are hopeful that the ideas here will be of interest to the reader, and perhaps these ideas will be used, in addition to some additional ideas, to obtain an improved tampering bound.

At first it may seem that it should be quite trivial to obtain a leakage and tamper resilient circuit, given that we can get leakage resilience and tamper resilience separately: First convert the circuit C_s to a leakage resilient circuit, and

[14] The choice of the constant η will depend on the parameters of the expander graph $\mathcal{G}_M$. Additionally, in what follows we assume the PCPP has (constant) soundness of at least $1-\eta/2$.

then apply our compiler from Section 4 to make it also tamper resilient. However, this approach does not necessarily work. For example, suppose the leakage-resilient circuit is secure as long as at most a certain fraction of the wires are leaked, as in the leakage model of Ishai *et. al.* [38]. After applying the compiler from Section 4 to the leakage-resilient circuit, the wire structure completely changes, and thus in the new circuit some wires may actually contain very sensitive information. Moreover, even if the resultant circuit is secure against leakage and tampering individually, it may not be secure against simulataneous leakage and tampering: The guaranteed leakage simulator for the resultant circuit cannot necessarily simulate leakage on a wire, w_j, after tampering has occurred on a previous wire, w_i.

We overcome this obstacle, by requiring that our leakage resilient circuit is secure against a more general class of leakage functions (as opposed to only leakage on wires). Specifically, we convert the original circuit C_s to a circuit that is secure against *arbitrary* (bounded) leakage in the *only computation leaks* (OCL) model. The OCL model, introduced by Micali and Reyzin [49], assumes that secret information that is merely stored in memory does not leak, but *any* information that is used during a computation may leak. Recently, it was shown how to convert any circuit into one that is secure in the OCL model [34, 39, 33]. Such a compiler is referred to as an OCL-compiler. In particular, the recent work of Goldwasser and Rothblum [33] constructs an OCL-compiler without relying on secure leak-free hardware, unlike the previous works.

Specifically, Goldwasser and Rothblum construct an efficient OCL-compiler that takes any circuit C_s and converts it into a leakage-resilient circuit, consisting of several *modules*, each of which performs a specific sub-computation. The security guarantee is that an adversary, who at any point of time throughout the computation may obtain an *arbitrary* bounded leakage from the "currently active" module, does not learn any more information than having black-box access to the circuit.

Our final compiler, which converts any circuit into a leakage and tamper resilient one, does not use the OCL-compiler, but in fact uses a variant of this compiler, known as an LDS compiler, introduced in the recent work of Bitansky *et. al.* [6] (where LDS stands for *leaky distributed systems*). The main difference between the LDS model and the OCL model is that in the LDS model the adversary can view the entire communication between the modules, and moreover, is allowed to feed the modules with inputs of his choice. Namely, the adversary can view and tamper arbitrarily with the messages sent between the modules. In other words, the LDS compiler generates a set of modules, and security is guaranteed against an adversary that is given these modules (which are thought of as leaky untamperable hardware), and can play with them by feeding them inputs of his choice, and by leaking arbitrary bounded functions on each module. For the sake of completeness, we give a formal definition of this model in Appendix A.

The high level idea of our compiler, which we denote by $\mathcal{LT}$ (for leakage and tampering), is the following: Given a circuit C_s first apply the LDS compiler to it, to obtain a collection of sub-computations (or "modules") $\mathsf{Sub}_1, \ldots, \mathsf{Sub}_k$, whose sequential evaluation evaluates the circuit C_s, and which is secure in the LDS model. Then, apply our compiler $\mathcal{T}$ from Section 4 to each sub-computation, to obtain a new collection of tamper resilient modules $\mathcal{T}(\mathsf{Sub}_1), \ldots, \mathcal{T}(\mathsf{Sub}_k)$.[15]

We emphasize that the modules $\mathcal{T}(\mathsf{Sub}_1), \ldots, \mathcal{T}(\mathsf{Sub}_k)$ remain secure against leakage. As discussed above, this would not necessarily be the case if the leakage model was weaker; say, if the leakage model only allowed leakage of wires. However, since the collection $(\mathsf{Sub}_1, \ldots, \mathsf{Sub}_k)$ is resilient to *arbitrary* (bounded) leakage on each Sub_i, the collection $\mathcal{T}(\mathsf{Sub}_1), \ldots, \mathcal{T}(\mathsf{Sub}_k)$ is also resilient to *arbitrary* (bounded) leakage on each $\mathcal{T}(\mathsf{Sub}_i)$. Loosely speaking, this is the case since any leakage $L(\mathcal{T}(\mathsf{Sub}_i))$ on the module $\mathcal{T}(\mathsf{Sub}_i)$ can be thought of as leakage $(L \circ \mathcal{T})(\mathsf{Sub}_i)$ on the module Sub_i.

In order to argue that $\mathcal{T}(\mathsf{Sub}_1), \ldots, \mathcal{T}(\mathsf{Sub}_k)$ is resilient to tampering, we need our "self-destruct" mechanism to destruct the secret states of *all* the modules. To this end, we change the gates G_{cas} of each module, so that if the output of G_{cas} is 0 in some module, then the secret states of *all* the modules are overwritten with 0.

We note that known LDS compilers have many modules (the number of modules is essentially the security parameter k), and thus even if each computation $\mathcal{T}(\mathsf{Sub}_i)$ can tolerate a constant fraction of tampering, the entire circuit $(\mathcal{T}(\mathsf{Sub}_1), \ldots, \mathcal{T}(\mathsf{Sub}_k))$ can tolerate at most $1/k$ fraction of tampering. Thus, by making our circuits resilient to leakage we pay in our tampering bounds, and go from constant tampering rate to $\mathrm{poly}(1/k)$ tampering rate.

We note that even if there did exist an LDS compiler with constant number of modules, our tampering rate will still reduce to $1/k$. One reason is that the number of bits sent from module i to module $i+1$ is essentially the security

[15] One can also apply the compiler from Section 5. However, the resulting compiler will be more complicated. Thus, for simplicity, we apply the compiler from Section 4.

parameter. Recall that in Section 4 we prove that our compiler converts circuits that output a *single* bit into circuits that are resilient to constant fraction of tampering. However, if the underlying circuit outputs ℓ bits, then the straightforward extension of our compiler, would yield a tamper-resilient circuit that is secure against $\mathrm{poly}(1/\ell)$-fraction of tampering. Moreover, a more inherent reason why we pay in the leakage rate stems from the fact that we wish to achieve security against *continual* leakage.

In order to achieve security against *continual* leakage, the secret states of the modules must be periodically updated; indeed, the OCL (or LDS) compilers include a randomized update procedure, denoted by Update, which the modules need to run in order to refresh (or update) their secret state (see Appendix A for details).

This raises a new technical difficulty in our setting: The (randomized) update procedure can be tampered with. In what follows we give a very high-level intuition of how one can overcome this difficulty. Suppose for now that the randomness used by the update procedure is not tampered with (we will deal with that later).

At first it may seem that to overcome this difficulty all we need to do is simply apply our tamper-resilient transformation and run a tamper-resilient update procedure $\mathcal{T}(\mathsf{Update})$. Unfortunately, this does not take us far, since the only security guarantee that $\mathcal{T}$ grants us is that the output can be simulated given black-box access to Update. This is not enough, since we need the guarantee that the update is done *correctly*, or at least "almost correctly". Therefore, instead, we make strong use of the fact that we only allow $1/\mathrm{poly}(k)$ of the wires to be tampered with. Namely, we run the (non tamper-resilient) update procedure Update, by using separate circuits to compute each output bit. Then we use a simple counting argument to argue that since we allow only $1/\mathrm{poly}(k)$ of the wires to be tampered with, most of the output bits are computed correctly. We note that we use here the fact that the size of the secret state of each module depends only on the security parameter k, and is independent of the circuit size. To guarantee that indeed the size of the modules and the number of modules, is independent of the circuit, the underlying LDS (or OCL) compiler uses a fully homomorphic encryption scheme.

If the randomness is not tampered with, then the above indeed seems to work. However, if the randomness is tampered with the update can fail completely.

We next deal with the fact that the randomness may be tampered with. To this end, each module $\mathcal{T}(\mathsf{Sub}_j)$, in addition to storing $\mathsf{ECC}(S_j)$ in memory (where S_j is the secret state of module j),[16] also stores $\mathsf{ECC}(v_j)$ in memory, where v_j is a random string. Namely, the secret state of module j is $(\mathsf{ECC}(S_j), \mathsf{ECC}(v_j))$. The idea is use $\mathsf{ECC}(v_j)$ in order to convert the possibly tampered random input r into an untampered random string.

More specifically, we run a modified update procedure, denoted by Update', which is defined as follows: On input two random strings (u, w) (that may be partially tampered with), it uses a 2-source extractor, Ext_2, to compute two (supposedly looking random) strings $v'_j = \mathsf{Ext}_2(v_j, u)$ and $r' = \mathsf{Ext}_2(v_j, w)$. Then, it computes and stores in memory both $\mathsf{ECC}(\mathsf{Update}(S_j; r'))$ and $\mathsf{ECC}(v'_j)$. These values overwrite the previous values $\mathsf{ECC}(S_j)$ and $\mathsf{ECC}(v_j)$, respectively. As mentioned above, the bits of $\mathsf{ECC}(\mathsf{Update}(S_j; r'))$ and $\mathsf{ECC}(v'_j)$ are generated using disjoint circuits so that we can argue most of these circuits are not tampered with.

Note that even if (u, w) and $\mathsf{ECC}(v_j)$ are partially leaked and partially tampered with, as long as they each still have some min-entropy left, then (v'_j, r') is close to uniform, assuming no leakage or tampering happens throughout the computation itself. We deal with tampering by simply assuming that most computations are not tampered with, since the tampering rate is $1/\mathrm{poly}(k)$. We deal with the leakage by assuming that the 2-source extractor Ext_2 is *inverse sampleable*, and thus the leakage can be simulated. We elaborate on this technicality in the proof sketch below.

We denote the resulting compiler by $\mathcal{LT}$, and give a high-level proof sketch that indeed $\mathcal{LT}$ converts any circuit into one that is secure against continual tampering (as defined in Section 3) and continual LDS-leakage (which is more general than OCL leakage, see Appendix A for the precise definition).

Informal Claim. The compiler $\mathcal{LT}$ (described above) converts any circuit into one that is secure against continual tampering (as defined in Section 3) and continual LDS-leakage (as defined in Appendix A).

Proof Sketch. Suppose there exists an adversary $\mathcal{A}$ that continually tampers and leaks from the circuit $\mathcal{LT}(C_s)$. We construct a simulator Sim that simulates the output of $\mathcal{A}$ given only black-box access to the circuit C_s.

[16] Recall that in Section 4.2, the secret state was $\widetilde{\mathsf{ECC}}(S_j)$. However, since in the leakage setting we only get tampering rate $1/\mathrm{poly}(k)$, we can use the simpler compiler from Section 4.1, which has $\mathsf{ECC}(S)$ as its secret state.

This is done in two steps: We first construct an adversary $\mathcal{A}'$ that simulates the output of $\mathcal{A}$ given black-box access to each of the modules $\mathsf{Sub}_1, \ldots, \mathsf{Sub}_k$, and given continual leakage on each of these modules. Then, we use the security in the LDS model (see Definition 7), to claim that there exists a simulator Sim who simulates the output of $\mathcal{A}'$ given only black-box access to the circuit.

Thus, we focus on constructing $\mathcal{A}'$, who internally emulates $\mathcal{A}$ as follows. The adversary $\mathcal{A}'$ first chooses random strings v_j for each module Sub_j, and thinks of the secret state of Sub_j as being $(\mathsf{ECC}(S_j), \mathsf{ECC}(v_j))$. Note that $\mathcal{A}'$ does not know $\mathsf{ECC}(S_j)$ but has a leakage oracle to it. The adversary $\mathcal{A}'$ emulates for $\mathcal{A}$ the output of each tampered run of the circuit, as follows:

Each time $\mathcal{A}$ runs the circuit on some input with some set of tampering instructions, the adversary $\mathcal{A}'$ simulates the output exactly as the simulator simulates these outputs in the proof of Theorem 3. Namely, Theorem 3 guarantees that $\mathcal{A}'$ can simulate the output of each module given only black-box access to the module,[17] and given one additional bit of leakage, which leaks whether the memory was self destructed already or not. Thus, $\mathcal{A}'$ will learn when a "self destruct" happens.

Note that during each such run the secret states of the modules may slightly change due to tampering. The adversary $\mathcal{A}'$ keeps track of the tampered secret states as a function of the original secret states. Namely, at any point, for every module $\mathcal{T}(\mathsf{Sub}_j)$, whose original secret state is $(\mathsf{ECC}(S_j), \mathsf{ECC}(v_j))$ (where S_j is the secret state of module j, and v_j is a random string), the adversary maintains a function T_j, such that the current secret state of module $\mathcal{T}(\mathsf{Sub}_j)$ is $T_j(\mathsf{ECC}(S_j), \mathsf{ECC}(v_j))$. Each time the adversary $\mathcal{A}$ requests a leakage query L on the j'th module, the adversary $\mathcal{A}'$ simulates the leakage to be $f(S_j) \triangleq L(T_j(\mathsf{ECC}(S_j), \mathsf{ECC}(v_j)))$. We use here the fact that the adversary $\mathcal{A}'$ knows v_j.

Recall that when self destruction occurs, all the secret states are overwritten with 0. However, since the adversary $\mathcal{A}$ may tamper with some of the wires that overwrite the memory, it may be the case that a few of the memory gates still contain some sensitive information. The adversary $\mathcal{A}'$ will leak all this sensitive information in each module. From now on, the simulation becomes trivial since $\mathcal{A}'$ has all the secret information.

It remains to show how $\mathcal{A}'$ emulates a run of the update procedure, before the memory is self destructed. Each time $\mathcal{A}$ requests a run of Update', the adversary $\mathcal{A}'$ requests a run of Update. As a result, the secret state of each module is updated to $S'_j = \mathsf{Update}(S_j; r_j)$. The adversary $\mathcal{A}'$ chooses at random u and updates $v'_j = \mathsf{Ext}_2(v_j, u)$. If the adversary $\mathcal{A}$ requests a leakage query L during the update procedure, the adversary $\mathcal{A}'$ uses his leakage oracle to simulate this leakage, as follows: It samples a random w' such that $r_j = \mathsf{Ext}_2(T_j(v_j), w')$, where we slightly abuse notation by denoting the outcome of the tampering function T_j restricted to v_j as $T_j(v_j)$. For this we need the 2-source extractor Ext_2 to be *inverse sampleable*; i.e., given the output r_j, and given a string $T_j(v_j)$, one can efficiently sample a random w', such that $r_j = \mathsf{Ext}_2(T_j(v_j), w')$. The inner product extractor is an example of an extractor that is inverse sampleable. Then $\mathcal{A}'$ lets w be a random string such that the tampering function T_j converts w to w'. Finally, $\mathcal{A}'$ can simulate leakage by requesting the leakage

$$f(S_j, r_j) \triangleq L(T_j(\mathsf{ECC}(S_j), \mathsf{ECC}(v_j)), u, w).$$

We note that during each run of the update procedure Update', the updated secret state of each module may slightly change due to tampering. The adversary $\mathcal{A}'$ keeps track of each tampered (updated) secret states as a function of the (updated) secret state.

References

1. Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *TCC*, pages 474–495, 2009.
2. Benny Applebaum, Danny Harnik, and Yuval Ishai. Semantic security under related-key attacks and applications. Cryptology ePrint Archive, Report 2010/544, 2010. http://eprint.iacr.org/.
3. Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: Rka-prps, rka-prfs, and applications. In *EUROCRYPT*, pages 491–506, 2003.
4. Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. Robust pcps of proximity, shorter pcps, and applications to coding. *SIAM J. Comput.*, 36(4):889–974, 2006.

[17] We note that we crucially rely here on the fact that $\mathcal{A}'$ has black-box access to each of the modules $\mathsf{Sub}_1, \ldots, \mathsf{Sub}_k$. This is why we need to use the LDS compiler, as opposed to the OCL compiler (see Appendix A for details).

5. Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO*, 1997.
6. Nir Bitansky, Ran Canetti, Shafi Goldwasser, Shai Halevi, Yael Tauman Kalai, and Guy N. Rothblum. Program obfuscation with leaky hardware. In *ASIACRYPT*, pages 722–739, 2011.
7. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In *EUROCRYPT*, pages 37–51, 1997.
8. Elette Boyle, Shafi Goldwasser, Abhishek Jain, and Yael Tauman Kalai. Multiparty computation secure against continual memory leakage. In *STOC*, 2012.
9. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *IACR Cryptology ePrint Archive*, 2011:277, 2011.
10. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *FOCS*, pages 97–106, 2011.
11. Ran Canetti, Yevgeniy Dodis, Shai Halevi, Eyal Kushilevitz, and Amit Sahai. Exposure-resilient functions and all-or-nothing transforms. In *EUROCRYPT*, pages 453–469, 2000.
12. Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *STOC*, pages 639–648, 1996.
13. Seung Geol Choi, Dana Dachman-Soled, Tal Malkin, and Hoeteck Wee. Improved non-committing encryption with applications to adaptively secure protocols. In *ASIACRYPT*, pages 287–302, 2009.
14. Seung Geol Choi, Aggelos Kiayias, and Tal Malkin. Bitr: Built-in tamper resilience. In *ASIACRYPT*, pages 740–758, 2011.
15. Ivan Damgård and Jesper Buus Nielsen. Improved non-committing encryption schemes based on a general complexity assumption. In *CRYPTO*, pages 432–450, 2000.
16. R. L. Dobrushin and S. I. Ortyukov. Upper bound for the redundancy of self- correcting arrangements of unreliable functional elements. 13:203–218, 1977.
17. Yevgeniy Dodis, Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Public-key encryption schemes with auxiliary inputs. In *TCC*, pages 361–381, 2010.
18. Yevgeniy Dodis, Yael Tauman Kalai, and Shachar Lovett. On cryptography with auxiliary input. In *STOC*, pages 621–630, 2009.
19. Yevgeniy Dodis and Krzysztof Pietrzak. Leakage-resilient pseudorandom functions and side-channel attacks on feistel networks. In *CRYPTO*, pages 21–40, 2010.
20. Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *FOCS*, pages 293–302, 2008.
21. Stefan Dziembowski, Krzysztof Pietrzak, and Daniel Wichs. Non-malleable codes. In *ICS*, pages 434–452, 2010.
22. W. Evans and L. Schulman. On the maximum tolerable noise of k-input gates for reliable computation by formulas.
23. W. Evans and L. Schulman. Signal propagation and noisy circuits. In *IEEE Trans. Inform. Theory, 45(7)*, pages 2367–2373, 1999.
24. Sebastian Faust, Eike Kiltz, Krzysztof Pietrzak, and Guy N. Rothblum. Leakage-resilient signatures. In *TCC*, pages 343–360, 2010.
25. Sebastian Faust, Krzysztof Pietrzak, and Daniele Venturi. Tamper-proof circuits: How to trade leakage for tamper-resilience. In *ICALP (1)*, pages 391–402, 2011.
26. 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.
27. T. Feder. Reliable computation by networks in the presence of noise. In *IEEE Trans. Inform. Theory, 35(3)*, pages 569–571, 1989.
28. Péter Gács and Anna Gál. Lower bounds for the complexity of reliable boolean circuits with noisy gates. *IEEE Transactions on Information Theory*, 40(2):579–583, 1994.
29. Anna Gál and Mario Szegedy. Fault tolerant circuits and probabilistically checkable proofs. In *Structure in Complexity Theory Conference*, pages 65–73, 1995.
30. Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *CHES*, number Generators, pages 251–261, 2001.
31. Rosario Gennaro, Anna Lysyanskaya, Tal Malkin, Silvio Micali, and Tal Rabin. Algorithmic tamper-proof (atp) security: Theoretical foundations for security against hardware tampering. In *TCC*, pages 258–277, 2004.
32. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.
33. Shafi Goldwasser and Guy Rothblum. How to compute in the presence of leakage. Electronic Colloquium on Computational Complexity, 2012.
34. Shafi Goldwasser and Guy N. Rothblum. Securing computation against continuous leakage. In *CRYPTO*, pages 59–79, 2010.
35. Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.
36. Bruce E. Hajek and Timothy Weller. On the maximum tolerable noise for reliable computation by formulas. *IEEE Transactions on Information Theory*, 37(2):388–, 1991.

37. Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and David Wagner. Private circuits ii: Keeping secrets in tamperable circuits. In *EUROCRYPT*, pages 308–327, 2006.
38. Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.
39. Ali Juma and Yevgeniy Vahlis. Protecting cryptographic keys against continual leakage. In *CRYPTO*, pages 41–58, 2010.
40. Yael Tauman Kalai, Bhavana Kanukurthi, and Amit Sahai. Cryptography with tamperable and leaky memory. In *CRYPTO*, pages 373–390, 2011.
41. Yael Tauman Kalai, Anup Rao, and Allison Lewko. Formulas resilient to short-circuit errors, 2011. Manuscript.
42. Jonathan Katz and Vinod Vaikuntanathan. Signature schemes with bounded leakage resilience. In *ASIACRYPT*, pages 703–720, 2009.
43. John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In *ESORICS*, pages 97–110, 1998.
44. Daniel J. Kleitman, Frank Thomson Leighton, and Yuan Ma. On the design of reliable boolean circuits that contain partially unreliable gates. In *FOCS*, pages 332–346, 1994.
45. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.
46. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.
47. Markus G. Kuhn and Ross J. Anderson. Soft tempest: Hidden data transmission using electromagnetic emanations. In *Information Hiding*, pages 124–142, 1998.
48. Feng-Hao Liu and Anna Lysyanskaya. Algorithmic tamper-proof security under probing attacks. In *SCN*, pages 106–120, 2010.
49. Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In *TCC*, pages 278–296, 2004.
50. Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. In *CRYPTO*, pages 18–35, 2009.
51. Krzysztof Pietrzak. A leakage-resilient mode of operation. In *EUROCRYPT*, pages 462–482, 2009.
52. N. Pippenger. Reliable computation by formulas in the presence of noise. In *IEEE Trans. Inform. Theory, 34(2)*, pages 194–197, 1988.
53. Nicholas Pippenger. On networks of noisy gates. In *FOCS*, pages 30–38. IEEE, 1985.
54. Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *E-smart*, pages 200–210, 2001.
55. Josyula R. Rao and Pankaj Rohatgi. Empowering side-channel attacks. Cryptology ePrint Archive, Report 2001/037, 2001. http://eprint.iacr.org/.
56. J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. 1956.

A Leaky Distributed Systems

In this section, we define the notion of security in the *leaky distributed systems* (LDS) model, as defined in the work of Bitansky *et. al.* [6]. To this end, we start by defining security in the *Only Computation Leaks* (OCL) model, as defined by Micali and Reyzin [49]. The text in this section is taken almost verbatim from [8].

We start by recalling the works [39, 34, 33], which construct a compiler that converts any circuit into one which is secure in the in the OCL model. These works construct a compiler that takes a circuit C and converts it into a circuit C' consisting of m disjoint, ordered sub-computations $\mathsf{Sub}_1, \ldots, \mathsf{Sub}_m$, where the input to sub-computation Sub_i depends only on the output of earlier sub-computations. Each of these sub-computations Sub_i is modeled as a non-uniform randomized poly-size circuit, with a "secret state." It was proven that no information about the circuit C is leaked, even if each of these sub-computations is leaky. More specifically, the adversary can request to see a bounded-length function of each Sub_i (separately), and these leakage functions may be adaptively chosen.

These works also consider the continual leakage setting, where leakage occurs over and over again in time. In this setting, the secret state of each Sub_i must be continually updated or refreshed. To this end, after each computation, all the Sub_i's update their secret state by running a randomized protocol Update. We stress that leakage may occur during each of these update protocols, and that such leakage may be a function of both the current secret state and the randomness used by the Update procedure.

The LDS model strengthens the OCL model in two ways. First, in the LDS model, the adversary is allowed to *view and control the entire communication between modules*; in contrast, the OCL model assumes the communication between modules is kept secret from the adversary, and that the messages are generated honestly. Second, in the LDS model, the adversary may leak adaptively on each module *in any order*. For instance, the adversary may leak a bit from Sub_i, then a bit from Sub_j, and based on the results, leak again on Sub_i. In contrast, the OCL model only allows the adversary to request leakage information from the module that is currently computing. In particular, this restricts the adversary to leak on modules in order (i.e., first leak from Sub_1, then from Sub_2, etc.).

In what follows, we describe the LDS compiler of [6] in more detail. Similar to the OCL circuit compiler, the LDS circuit compiler initializes each module with some secret state, and thereafter the modules can receive messages, send messages, generate fresh randomness, and maintain a local state. To evaluate the circuit C, an input v is given to Sub_1 and the modules communicate to jointly compute $C(v)$, which is eventually output by Sub_m.

Remark 1. For the sake of simplicity of notation, we assume (without loss of generality) that the module Sub_i only sends messages to Sub_{i+1} (where we define $\mathsf{Sub}_{m+1} \triangleq \mathsf{Sub}_1$). Moreover, we assume for simplicity that during each computation, where C is evaluated on some input v, each module Sub_i sends a *single* message to Sub_{i+1}, and that Sub_m does not send a message to any module, and simply outputs $C(v)$. This assumption indeed holds for the LDS compiler of [6] which is based on [33]. We note that this assumption is not needed for our result to be correct, but it simplifies the notation.

The LDS model considers an adversary who interacts with and leaks from the modules in attempt to learn C. As in the OCL model, the adversary is allowed to freely choose the inputs v to the computation, and to choose one submodule at a time and evaluate a leakage function on its inner state and randomness. There is no limit to the number of times each component can be chosen to leak during the lifetime of the system, as long as the total rate of leakage from each component is not too high. In addition, the leakage functions can be chosen adaptively, as a function of all information learned up to that point. However, in contrast to the OCL model, the adversary is allowed to leak on the modules in any order. The adversary also has the additional power to see and control all the communication between the modules.

More explicitly, the LDS model considers continual leakage, where the lifetime of each submodule is partitioned into time periods. At the end of each time period, the modules "refresh" their inner state by applying a (possibly distributed) Update procedure, after which they erase their previous state. As with the rest of the computation, the Update procedure is also exposed to leakage, and the adversary controls the exchange of messages during the update.

Definition 6 (Leaky Distributed Systems (LDS) Model). *In a λ-bounded* LDS attack*, a $\mathcal{PPT}$ adversary $\mathcal{A}$ interacts with modules* $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$ *by adaptively performing any sequence of the following actions:*

- $\mathsf{Interact}(j, \mathsf{msg})$**:** *For $j \in [m]$, send the message* msg *to the j'th submodule, Sub_j, and receive the corresponding reply. Note that the modules are message-driven: they become activated when they receive a message from the attacker, at which point they compute and send the result, and then wait for additional messages.*
- $\mathsf{Leak}(j, L)$**:** *For $j \in [m]$ and a poly-size leakage function $L : \{0,1\}^* \to \{0,1\}$, if strictly fewer than λ queries of the form $\mathsf{Leak}(j, \cdot)$ have been made so far, $\mathcal{A}$ receives the evaluation of L on the secret state of the j'th submodule, Sub_j. Otherwise, $\mathcal{A}$ receives $\perp$.*

In a continual λ-LDS attack*, the adversary $\mathcal{A}$ repeats a λ-bounded* LDS *attack polynomially many times, where between every two consecutive attacks the secret states of the modules are updated. The update is done by running a distributed* Update *protocol among all the modules. We also allow $\mathcal{A}$ to leak during the* Update *procedure, where the leakage function takes as input both the current secret state of* Sub_j *and the randomness it uses during the* Update *procedure.*

We denote by time period t of submodule Sub_j *the time period between the beginning of the $(t-1)$'st* Update *procedure and the end of the t'th* Update *procedure in that submodule (note that these time periods are overlapping).*[18] *We allow the adversary $\mathcal{A}$ to leak at most λ bits from each* Sub_j *during each (local) time period.*

*We refer to such an adversary $\mathcal{A}$ as an λ-*LDS adversary*, and denote the output of $\mathcal{A}$ in such an attack by* $\mathcal{A}[\lambda : \mathsf{Sub}_1, ..., \mathsf{Sub}_m : \mathsf{Update}]$.

We say that the collection of modules $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$ is λ-*secure* in the LDS model if for any λ-LDS adversary $\mathcal{A}$ interacting with the modules as described above, there exists a $\mathcal{PPT}$ simulator who simulates the output of $\mathcal{A}$.

[18] Intuitively, time period t is the entire time period where the t'th updated secret states can be leaked. Note that during the t'th Update procedure, both the $(t-1)$'st and the t'th secret state may leak, which is why the time periods are overlapping.

Definition 7 (LDS**-Secure Circuit Compiler).** *We say that* $(\mathsf{C}, \mathsf{Update})$ *is a* λ-LDS secure circuit compiler *if for any circuit* C *and* $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m) \leftarrow \mathsf{C}(C)$*, the following two properties hold:*

1. **Correctness:** *The collection of modules* $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$ *maintain the functionality of* C *when all the messages between them are delivered intact.*
2. **Secrecy:** *For every* $\mathcal{PPT}$ λ-LDS *adversary* $\mathcal{A}$ *there exists a* $\mathcal{PPT}$ *simulator* Sim*, such that for any ensemble of poly-size circuits* $\{C_n\}$ *and any auxiliary input* $z \in \{0,1\}^{\mathrm{poly}(n)}$*:*

$$\left\{\mathcal{A}(z)[\lambda : \mathsf{Sub}_1, ..., \mathsf{Sub}_m : \mathsf{Update}]\right\}_{n\in\mathcal{N}, C\in C_n} \approx_c \left\{\mathsf{Sim}^C(z, 1^{|C|})\right\}_{n\in\mathcal{N}, C\in C_n},$$

where Sim *only queries* C *on the inputs* $\mathcal{A}$ *sends to the first module,* Sub_1.

Theorem 4 **([6]).** *Assuming the existence of a non-committing encryption scheme and a* λ-OCL *circuit compiler which compiles a circuit* C *to* $m(|C|)$ *modules, there exists a* λ-LDS *secure circuit compiler* $(\mathsf{C}, \mathsf{Update})$ *for which* $\mathsf{C}(C)$ *has the same number of modules,* $m(|C|)$.

A very recent work of Goldwasser and Rothblum [33] constructs a λ-OCL circuit compiler with the following properties.

Theorem 5 **([33]).** *For any security parameter* k*, there (unconditionally) exists a* λ-OCL *secure circuit compiler for* $\lambda = \tilde{\Omega}(k)$*, that takes any circuit* C *into a collection of* $O(|C|)$ *modules, each of size* $O(k^3)$.

Remark 2 (Folklore). If one additionally assumes the existence of a fully homomorphic encryption (FHE) scheme, then there exists a λ-LDS secure circuit compiler $(\mathsf{C}, \mathsf{Update})$ such that for every poly-size circuit C, the number of output sub-computations $\mathsf{Sub}_1, ..., \mathsf{Sub}_m$ generated by C is polynomial in the security parameter of the FHE scheme and *independent* of the size of C.