Securing Computation Against Continuous Leakage

Shafi Goldwasser[1,⋆] and Guy N. Rothblum[2,⋆⋆]

[1] Weizmann Institute of Science and MIT
[2] Princeton University

Abstract. We present a general method to compile any cryptographic algorithm into one which resists side channel attacks of the *only computation leaks information* variety for an unbounded number of executions. Our method uses as a building block a semantically secure subsidiary bit encryption scheme with the following additional operations: key refreshing, oblivious generation of cipher texts, leakage resilience re-generation, and blinded homomorphic evaluation of one single complete gate (e.g. NAND). Furthermore, the security properties of the subsidiary encryption scheme should withstand bounded leakage incurred while performing each of the above operations.

We show how to implement such a subsidiary encryption scheme under the DDH intractability assumption and the existence of a simple secure hardware component. The hardware component is independent of the encryption scheme secret key. The subsidiary encryption scheme resists leakage attacks where the leakage is computable in polynomial time and of length bounded by a constant fraction of the security parameter.

1 Introduction

Modern cryptographic algorithms are designed under the assumption that keys are perfectly secret, and computations done within one's computer are opaque to the outside. Still, in practice, keys do get compromised at times, and computations are not fully opaque for a variety or reasons. A particularly disturbing loss of secrecy is as a result of *side channel attacks*.

These attacks exploit the fact that every cryptographic algorithm is ultimately implemented on a physical device and such implementations enable "observations" that can be made and measured on computations which use secret data and secret keys, or on the secret keys and data directly. Such observations can and have lead to complete breaks of systems which were proven secure, without violating any of the underlying mathematical principles. (see [KJJ99,RCL] for just two examples). Recently, a growing body of research on *side-channel-resilient cryptography* aims to build general mathematical models of realistic side

[⋆] This research is supported in part by ISF710267, BSF710613, NSF6914349 and an internal Weizmann KAMAR grant.

[⋆⋆] Research supported by NSF Grants CCF-0635297, CCF-0832797 and by a Computing Innovation Fellowship.

channel attacks, and to develop methods grounded in modern cryptography to provably resist these attacks.

Modeling side channel attacks on a cryptographic algorithm so as to simultaneously capture real world attacks and achieve the right level of theoretical abstraction, is an intriguing and generally controversial question. Indeed, the number of answers seems to be nearly as high as the number of papers published on the topic. Perhaps the only universally agreed on part of the modeling is that each physical measurement should be modeled as the result of computing an adversarially chosen but computationally bounded function ℓ (the so called "leakage" function) on the "internal state of the computation". We find the most important modeling questions to be:

- *How should we characterize which leakage functions ℓ can be measured?*
- *How many measurements occur and how often?*
- *Are all portions of the computation's internal state subject to measurement at the same time? Namely, what is the input to the leakage function ℓ?*
- *Can we use secure hardware components, and if so which ones are reasonable to assume as building blocks to achieve side channel security?*

"Only computation leaks information" and the question of granularity. Micali and Reyzin, in their pioneering work [MR04], set forth a model of physical security, which takes a particular approach at these modeling questions. One of the axioms in their model was that any computation *but only computation* leaks information (OC attack model). In other words, every time a computation step of a cryptographic algorithm "touches" data which may contain portions of (but not necessarily the entirety of): cryptographic secret keys, internally generated randomness, and results of previous computations done on cryptographic keys, a measurement on this data can be made by an adversary. However, data which is not "touched" by a computation step of an algorithm, can not be measured at this time (and thus does not leak). Stated in terms of leakage functions, this means that a leakage function can be computed in each computation step, but each such function is restricted to operate only on the data utilized in that computation step. Within this model, various constructions of cryptographic primitives [GKR08,DP08,Pie09,FKPR09] such as stream ciphers and digital signatures, have been proposed and proved secure for certain leakage function classes and under various computational intractability assumptions.

This is the model of attacks which we focus on in this paper. Our main result addresses *how to run any cryptographic algorithm* (i.e an algorithm which takes as input secret keys and uses secret randomness) securely in this model for an unbounded number of executions.

Implicit in using this model, is the view of program execution (or computation) as preceding in discrete 'sub-computation steps' S_1, S_2... Each subcomputation S_i computes on some data d_i (which is a combination of secret and public data and randomness). At each S_i, the side-channel attack adversary can request to receive the evaluation of a new leakage function ℓ_i on d_i. The choice of ℓ_i to be evaluated at step S_i may depend on the results of values attained by previous $\ell_1, ..., \ell_{i-1}$, but ℓ_i can only be evaluated on the d_i used in step S_i.

An important question in evaluating results in the OC attack model emerges: what constitutes a sub-computation step S_i, or more importantly what is the input data d_i to S_i available to ℓ_i in this sub-computation? Let us look for example at the beautiful work of Dziembowski and Pietrzak [DP08] which construct secure *stream ciphers* in the OC model. Initialized with a secret key, their stream cipher can produce an unbounded number of output blocks. In [DP08], the i-th sub-computation is naturally identified with the computation of the i-th block of the stream cipher. The input d_i to this sub-computation includes a pre-defined function of the original input secret key. The class of tolerated leakage functions ℓ_i (each computed on d_i) are (roughly) restricted to a length shrinking function whose output size is logarithmic in the size of the security parameter of the stream cipher[3]. Another example is in the work of Faust *et al.* [FKPR09] which construct secure randomized digital signature scheme which can generate an unbounded number of signatures in the OC attack model. The i-th sub-computation is identified with the computation of the ith signature, and d_i is (essentially) fresh randomness generated for the i-th sub-computation. Coupled with one-time signatures of [KV09], the class of leakage functions ℓ_i tolerated are length shrinking functions whose output size is a constant fraction of the size of the security parameter of the signature scheme, under the intractability of DDH and various lattice problems.

An interesting practical as well as theoretical question is what *granularity (i.e. size of sub-computations)* is reasonable to consider for general cryptographic computation. Certainly, the larger the granularity (and the sub-computations), the better the security guarantee. For security, ideally we'd prefer to allow the leakage to work on the entire memory space of the computation. However, the assumption that physical leakage is "local" in time and space, and applies to small sub-computations as they happen, still encapsulates a rich family of attacks. Carried to the extreme, one might even model leakage as occurring on every single gate of a physical computation with some small probability, and even this model may be interesting.

In this work, we advocate the approach of allowing the programmer of a cryptographic computation, the freedom to divide the computation into arbitrary sub-computations, and then analyzing security by assuming that leakage is applied to each sub-computation's input independently (i.e. only computation leaks information). In particular, this will mean that the total amount of leakage from a computation can grow with the complexity of the computation (as well as the number of executions), as it well should, since indeed in practice the possibility of leakage increases with the complexity (length of time) of the computation. General approach aside, our positive results are much stronger: we work with granularity that is a polynomial in a security parameter.

[3] Alternatively stated, their construction of is based on an exponential hardness assumption (where the assumption degrades as a function of the amount of leakage tolerated).

1.1 The Contributions of this Work

In this work we focus on general cryptographic computations in the OC attack mode, and address the challenge of how to run *any cryptographic algorithm* securely under this attack model, for any polynomial number of executions.

Our contributions are twofold. First, we show a reduction. Starting with a subsidiary semantically secure bit encryption scheme E, which obeys certain additional homomorphic and leakage-resilience properties (see below and Section 3), we build a compiler that takes any cryptographic algorithm in the form of a Boolean circuit, and probabilistically transforms it into a functionally equivalent probabilistic stateful algorithm. The produced algorithm can be run by a user securely for an unbounded number of executions in the presence of continuous OC side-channel attacks. Second, we show how to implement such a subsidiary encryption scheme E under the DDH intractability assumption and using a secure hardware component. The hardware component (see Section 1.1) samples from fixed polynomial time computable distribution (it does not compute on any secrets of the computation). The security assumed about the component is that there is no leakage on the randomness it uses or on its inner workings.

The execution and adversary model: We start with a cryptographic algorithm C and its secret key y (C is a member of a family of poly(n)-size Boolean circuits $\{C_n\}$ and $y \in \{0,1\}^n$). In an initial *off-line stage when no side-channel attacks are possible*, $C(y,\cdot)$ is converted via a probabilistic transformation to an algorithm $Eval_C$ with state – which is updated each time $Eval_C$ is executed, and is functionally equivalent to C, i.e $Eval_C(\cdot) = C(y,\cdot)$. After this initial off-line stage, $Eval_C$ is computed on an unbounded number of public inputs x_1, x_2,... which can be chosen by the adversary in the following manner. The computation of $Eval_C(x_i)$ is divided into sub-computations $C_{i,1},...,C_{i,n}$ each of which are evaluated on data $d_{i,1},...,d_{i,n}$ respectively. At this stage, for each sub-computation $C_{i,j}$, the OC side-channel adversary is allowed to request the result of evaluating leakage function $\ell_{i,j}$ on $d_{i,j}$. The leakage functions we tolerate can be chosen adaptively based on the result of previously evaluated leakage functions, and belong to the class of polynomial time computable length shrinking functions. We emphasize that after the initial off-line stages all computations of $Eval_C$ (including its state update) are subject to OC side-channel attacks.

The security guarantee: is that even under the OC side-channels and adversarially chosen inputs, the adversary learns no more than the outputs of $C(y,\cdot)$ on the chosen inputs (formally, there is a simulation guarantee). In particular, it is important to distinguish between leakage incurred on the cryptographic algorithm $C(y,\cdot)$ being protected, and the leakage on the subsidiary cryptographic scheme. There is constantly leakage on the subsidiary scheme's secret keys, and the specific scheme we use can handle this. On the other hand, for the algorithm $C(y,\cdot)$ *there is no leakage at all on y.* Only its black-box behavior it exposed.

For example, if we think of C as the decryption algorithm for *any* public-key scheme, and y as its secret decryption key (which is completely unrelated to the secret keys of the subsidiary cryptosystem we use as a tool!), then an adversary who wants to decrypt a challenge ciphertext x_1, and has OC leakage access to the evaluation of $Eval_C(y, x_2)$ for decrypting another ciphertext x_2, still cannot break the security of x_1 and in particular cannot decrypt it. This is a qualitatively different security guarantee from the setting of memory-bound leakage [AGV09] or even in the more recent work of Brakerski *et al.* [BKKV10] on public key encryption under continual leakage. In these works, no security is guaranteed for a challenge ciphertext that is known to the adversary when it chooses the leakage function.

The granularity of our sub-computations: We let a subsidiary security parameter λ govern the granularity of the computation steps as follows. The computation of $Eval_C$ is divided into sub-computations each of which consist of performing a basic cryptographic operations (e.g. encrypt, decrypt, key generate, etc.) of a subsidiary encryption scheme E with security parameter λ. Essentially, E is used as a tool to emulate a secure executions of C, in such a way that a constant number of cryptographic operations of E emulate the evaluation of each gate of C. Thus the complexity of $Eval_C$ is $O(poly(\lambda) \cdot |C|)$. In accordance with the OC attack model, leakage functions are assumed to apply to each input of the cryptographic operations of E separately. The main idea behind obtaining the leakage resilience for any algorithm C, is that whereas how C works is out of our control (as it is a given), we can choose an E for which we are able to continually refresh its keys. As each key will be utilized as input for only a constant number of cryptographic operations, only a bounded number of leakage functions (measurements) can be made on each key. Indeed, for an appropriately chosen E, we can tolerate any polynomial time computable leakage functions, whose output length is up to a constant fraction of the security parameter λ. Note that the security parameter λ is chosen for the purposes of side-channel security, and may be chosen to be different than the security parameter n of the cryptographic algorithm C, by the implementer.

Leakage grows with the complexity of $Eval_C$: The total amount of leakage that our method can tolerate per execution of $Eval_C$ is $O(\lambda \cdot |C|)$ whereas and the complexity of $Eval_C$ is $O(poly(\lambda) \cdot |C|))$. Thus, our method tolerates more measurements and leakage as the computation time increases. This is in contrast with previous general compilers (see Section 1.2), where the size of the transformed circuit grows as a function of the total amount of leakage tolerated.

Main tool: a subsidiary cryptosystem The subsidiary cryptosystem utilized by our compiler is a semantically secure bit encryption scheme with the following special properties (even in the presence of OC side channel attacks). See Section 3 for full definitions of these properties.

• **Semantic Security under Multi-source Leakage.** We require semantic security to hold even against an adversary who can (measure) receive leakage both from the secret key *and the cipher-texts which we attempt to protect, and are encrypted under this secret key.* Note that we depart here from the [AGV09] model in considering leakage also on the challenge ciphertexts, and not only on the keys. A priori, this might seem impossible. The reason it is facilitated is that due to the OC nature of our attacks an adversary can never apply a leakage function to the ciphertext and the secret-key at the same time (otherwise it could decrypt); furthermore the leakage length bound ensures that the adversary will not learn enough of the ciphertext to be useful for him at a later time when it can apply an adaptively chosen leakage function to the secret key (otherwise, again, it could decrypt the ciphertext).

• **Key Refreshing.** It should be possible to "refresh" secret keys in the scheme, changing them into new keys, via a randomly generated *correlation value*. In addition, we require that using the correlation value alone and *without knowledge of the secret key*, one can also refresh old ciphertexts under the old secret key to new ciphertext under the new secret key. Intuitively, this property is useful for taking secret keys on which there has already been a large amount of leakage, and transforming them into new keys on which there is less leakage (i.e. with more entropy). The requirement that refreshing on ciphertexts must not use the secret key, is due to the fact that otherwise a leakage function could be evaluated on the ciphertext and key (which are computed on at the same time) simultaneously and used to decrypt the ciphertext! The fact that the correlation value alone can be used to refresh ciphertexts avoids attacks of this type.

• **Oblivious Ciphertext Generation.** It should be possible to generate fresh encryptions of random bits. Even an OC adversary should not be able to tell anything about the plaintexts in these new obliviously generated ciphertexts. For example, the Goldwasser-Micali [GM84] cryptosystem naturally has this property (by generating a random Jacobi symbol 1 element).

• **Leakage Resilience Regeneration.** It should be possible to "re-generate" leakage resilience on ciphertexts and keys: i.e., to take a ciphertext and secret key and repeatedly generate a new "random-looking" ciphertext and key pair, encrypting the same value. The security requirement is that even after many such regenerations (with accumulated ciphertext and key OC leakages), as long as the amount of leakage between two successive regenerations is bounded, an adversary cannot tell whether the original ciphertext was an encryption of 0 or of 1. Intuitively, this property is useful for taking old ciphertexts and keys, on which there has been previous leakage, and re-generating them into new ones that are more secure (i.e. injecting new entropy).

• **Blind Homomorphic NAND.** It should be possible to take three ciphertexts c_i, c_j, c_k, encryptions of b_i, b_j, b_k (respectively), and output a data string hc (a "homomorphic ciphertext") which can later be decrypted (using the secret key) to yield $b = (b_i \text{ NAND } b_j) \oplus b_k$.[4] Moreover, we require a "blinding" property: that the encrypted outcome hc contains no more information beyond the plaintext outcome b, even w.r.t an adversary who can launch OC attacks on the homomorphic evaluation, and who is later given OC access to the decryption of hc (which also computes on the secret key). In particular, such an adversary should not be able to learn anything about the values of b_i, b_j, b_k beyond b. Note that we do not require that hc itself be a ciphertext or support any further homomorphic operations: i.e. we require only "one-shot" homomorphism.

[4] Actually, we require homomorphic evaluation of a slightly more complex functionality that also takes some plain-text inputs, see Section 3.

Instantiating the subsidiary cryptosystem A slight modification of the encryption scheme of Naor and Segev [NS09] and Boneh et al[BHHO08], amplified with a simple secure hardware device, satisfies all of these properties. Here we highlight some of the novel challenges and ideas. See Section 3 for details.

We do not specify the scheme [NS09] in its full detail here, but only recall that operates over a group G of order q where the Decisional Diffie Hellman Problem (DDH) is hard. The secret key is a vector $\boldsymbol{s} \in \mathbb{GF}[\mathsf{q}]^m$ for some small $m > 0$ (for our parameters $m = 10$ suffices) and the public key is g^s, g for a generator g. To encrypt $b \in \mathbb{GF}[\mathsf{q}]$, the scheme masks g^b by multiplying it by a group element whose distribution is indistinguishable (under DDH) from $g^{\langle \boldsymbol{s},\boldsymbol{r} \rangle}$, where $\boldsymbol{r} \in \mathbb{GF}[\mathsf{q}]^m$ is uniformly random. We note further that the scheme supports homomorphic addition (over $\mathbb{GF}[\mathsf{q}]$) and scalar multiplication.

Semantic security under multi-source leakage. We need to prove that semantic security holds when an adversary can launch a "multi-source" leakage attack *separately* on the secret key and the cipher-texts which we attempt to protect (encrypted under this secret key), a (constant fraction) of leakage is computed on each. The proof of security uses ideas from theory of two source extractors. In particular a theorem of Chor and Goldreich [CG88], showing how to extract statistically close to uniform bits from two independent min-entropy sources. We argue (assuming DDH) that the adversary's view is indistinguishable from an attack in which the plaintext b is masked by $g^{\langle \boldsymbol{s},\boldsymbol{r} \rangle}$, where $\boldsymbol{r}$ is a uniformly random vector in $\mathbb{GF}[\mathsf{q}]^m$. Given the adversary's separate leakage functions from the key $\boldsymbol{s}$ and the ciphertext $\boldsymbol{r}$, $\boldsymbol{s}$ and $\boldsymbol{r}$ will have sufficient *entropy* (because the amount of leakage is bounded) and are *independent* random sources (because the leakage operates separately on key and ciphertext). Using [CG88] we conclude that $\langle \boldsymbol{s}, \boldsymbol{r} \rangle$, and also $g^{\langle \boldsymbol{s},\boldsymbol{r} \rangle}$, are statistically close to uniform. This is all in an attack where $\boldsymbol{r}$ is uniformly random, but this attack is (under DDH) indistinguishable from the real one, and so semantic security holds. No secure hardware is used here.

Key refresh. Key refresh is enabled by the homomorphic properties of the Naor-Segev cryptosystem. In particular, choosing a correlation value $\pi \in GFq^m$, we can add this value to the secret key and update the public key and any ciphertext accordingly in a homomorphic manner, without accessing the secret key. No secure hardware is used here.

Secure hardware. The secure hardware device **CipherGen** (see Section 3.1) that is used in this work is simple. The device receives as input a public key and mode of operation $mode \in \{0, rand\}$. In mode $mode = 0$ it computes and outputs a fresh encryption of 0, and in mode $mode = rand$ it chooses a uniformly random bit $b \in \{0, 1\}$ and computes and outputs a fresh encryption of b. I.e. it only runs public key operations. We assume that when this device is invoked, there is leakage on its input and output, but not on its internal workings or randomness. It is interesting to compare this device to the device used by Faust *et al.* [FRR$^+$09]. Their device samples a random string whose XOR is 0. This can be viewed as a string "encrypting" the bit 0. The adversary, who is bounded to AC^0

bounded length leakage functions, cannot determine the XOR, or "decryption", of the string that was generated. We also note that in several works addressing continual leakage for particular functionalities, it is assumed that during parts of the computation either there is *no leakage* from the computation's internal randomness [DHLAW10], or that leakage from the internal randomness is very limited [BKKV10].

Oblivious Generation and Leakage-Resilience Regeneration. These two properties are satisfied almost immediately by the **CipherGen** secure hardware device. Activating the device in mode *rand* generates opaquely a ciphertext encrypting a random plaintext bit — giving immediately an oblivious generation procedure. For ciphertext and key regeneration we first use key refreshing to regenerate the secret key (injecting new entropy). We then use mode 0 of **CipherGen** to generate a fresh encryption of 0, and add it to the ciphertext. This effectively regenerates the randomness of the ciphertext, injecting new entropy.

Homomorphic blinded masked NAND. Perhaps the most challenging obstacle in constructing the subsidiary cryptoscheme is coming up with a procedure for computing blinded homomorphic masked NAND, i.e. given ciphertext c_1, c_2, c_3 encrypting plaintexts $b_1, b_2, b_3 \in \{0, 1\}$, computing a homomorphic blinded ciphertext containing (b_1 NAND b_2) $\oplus$ b_3.

Suppose for a moment that $b_3 = 0$ (i.e. we are computing the NAND of b_1 and b_2). We could homomorphically add the three ciphertexts, obtaining an encryption d of a plaintext γ, where γ is either 0,1 or 2 (it is important the homomorphic addition here is over $\mathbb{GF}[\mathsf{q}]$ only). Here $\gamma = 2$ means that $b_1 = b_2 = 1$ and the NAND is 0, and $\gamma \in \{0, 1\}$ outcomes imply that the NAND is 1. We note however that the exact value of $\gamma \in \{0, 1\}$ leaks information about the input b_1 and b_2, which we will need to "blind".

There are two main ideas in blinding. The first is to use mode *rand* of **CipherGen** to generate an encryption u of a random bit in $\mu \in \{0, 1\}$. We can then homomorphically compute an encryption of $\gamma - \mu - 2$, which will always be non-zero if the NAND is 1, and will be zero w.p. 1/2 (over the ciphertext generated by **CipherGen**) if the NAND is 0. Similarly, for the case where $b_3 = 1$ we can compute an encryption of $\gamma - \mu$ which will have the same distribution depending on the value of the masked NAND. In conclusion, if we compute homomorphically an encryption of $b_1 + b_2 - \mu - 2 \cdot (1 - b_3)$ we obtain an encryption of a non-zero value when the NAND is 1, or a zero value w.p. 1/2 when the NAND is 0. Repeating this several times, for different u, all the homomorphic decryptor needs to do is check whether any of these homomorphic computations resulted in a zero plaintext (in which case the output is 0) or not (there is a negligible error probability of incorrect decryption). We emphasize, that for each ciphertext generated in the above procedure, being an encryption of a zero or non-zero plaintext exposes *no information about the inputs* (beyond the output). This is because even an OC leakage adversary cannot tell whether **CipherGen** generated an encryption of 0 or 1. In a different context and cryptosystem, similar ideas for blinding were used by Sander, Young and Yung [SYY99].

Still, another idea is necessary, as the specific non-zero plaintext value (e.g. 1 rather than 2) might leak information about the inputs. An initial observation is that homomorphic multiplication by a random scalar e leaves zero ciphertexts as encryptions of zero, but completely randomizes the plaintext values of non-zero ciphertexts. This can blind the ciphertexts while maintaining (for correctness) their plaintext being zero or non-zero (respectively). Unfortunately, in the presence of OC leakage there will be leakage on the value e, and this blinding will not be secure. We handle the OC leakage using a more complicated blinding procedure, which essentially homomorphically multiplies the plaintext by an inner product of two vectors $\boldsymbol{e}$ and $\boldsymbol{f}$ of random scalars. We use ciphertext regeneration (mode 0 of **CipherGen**) in-between homomorphic sub-steps to ensure that the leakage from each scalar is independent (or rather indistinguishable under DDH from an experiment with independent leakage). In the end, even given the leakage, the scalar $\langle \boldsymbol{e}, \boldsymbol{f} \rangle$ by which we multiply the ciphertext is indistinguishable from uniform, even to an OC leakage adversary, and blinding is guaranteed.

Main result: the compiler The main contribution of this paper is a compiler which takes any cryptographic algorithm in the form of a Boolean circuit, and transforms it into a functionally equivalent probabilistic stateful algorithm. In this overview we assume an intuitive understanding of the subsidiary encryption scheme E and its properties and letting (pk_j, sk_j) denote public and secret key pairs of E. See Section 4 for details. We emphasize that in the description that ensues there is a distinction between a user who is executing the evaluation algorithm and an adversary whose view of this execution (which proceeds by a sequence of sub-computations) is only through the results of leakage functions applied on secret data as sub-computations are actually performed on this data.

The *input* to the compiler is a *secret* input $y \in \{0,1\}^n$, and a *public circuit* C of size $poly(n)$ that is known to all (compiler and adversary alike). The circuit takes as inputs y and also public input $x \in \{0,1\}^n$ (which may have been chosen by the adversary), and produces a single bit output.[5] Without loss of generality, the circuit C is composed of NAND gates with fan-in and fan-out 2, which are organized in layers. The inputs of layer i arrive from the outputs of layer $i-1$. The *output* of the compiler on C and y is a probabilistic evaluation algorithm $Eval_C$ with state (which will be updated during the run of $Eval_C$) such that for all x, $C(y,x) = Eval_C(x)$. The compiler is run once at the beginning of time and is not subject to side-channels. See Section 2.2 for a formal security definition.

The idea of the evaluation algorithm is that in its state it keeps the value v_j of each wire j of the original input circuit $C(y,x)$ in the following secret-shared form: $v_j = a_j \bigoplus b_j$. The invariant for every wire is that the a_j shares are public and known to all whereas b_j are secret and kept encrypted by the subsidiary encryption algorithm E under a secret key sk_j (i.e. there is a key-pair for every wire). We emphasize that the OC side-channel adversary does not actually ever see even the cipher-text of plain text b_j – let alone b_j itself – in their entirely, but rather only the result of a leakage function on these cipher-texts at the time when they are involved in a sub-computation.

[5] We focus on single bit outputs, the case of multi-bit outputs also follows naturally.

At the outset of computing $Eval_C$, for all input wires corresponding to the y-input, $a_j = 0$; for all input wires corresponding to the x input, $b_j = 0$; for all the other wires b_j are chosen uniformly at random independently of the input; This generation of random ciphertexts containing the b_j value is done using the oblivious generation procedure of E. Finally, for the circuit's output wire, $b_{output} = 0$. As the user selects an input x to run $Eval_C$ on, he sets a_j on the input wires of the x-input by the value of the bits of x, and is now ready to start updating the shares a_j on the internal wires and compute $a_{output} = C(y, x)$.

The crux of the idea is to show how the user can compute the public shares corresponding to the internal wires of $C(y, x)$. Here is where we use the fact that encryption scheme E can support a blinded homomorphic evaluation of a single NAND gate. Say, the user already computed the values of a_j of all wires j on layer s (starting from the input wires this will hold inductively). Then, for each pair of wires i, j into a gate on layer $s + 1$ with output wire k, the user will compute the public share of the output wire a_k via a sequence of sub-computations as follows: first, transform the cipher texts of b_i, b_j (using the key-refresh property) to encryptions of the same plaintexts under the secret key pk_k; second, homomorphically using a_i, a_j and the cipher texts of b_i, b_j, b_k all under pk_k compute a (blinded) ciphertext hc_k of a_k under pk_k (note that $a_k = ((a_i \oplus b_i) \text{ NAND } (a_j \oplus b_j)) \oplus b_k)$ [6] and finally, decrypt the blinded hc_k using secret key sk_k to obtain a_k. Note that this is one place the "only computation leaks information" assumption is of essence. For example, if the leakage function would have taken the inputs to the first sub-computation as well as to the third sub-computation, it could have used sk_k to decrypt b_i and discover in full the value of v_i, which of course would destroy the security of the entire construction (since it is non black-box information about the computation being performed). It is also important to note here that we will set the leakage parameter λ to be such that the adversary cannot even see enough of the ciphertexts corresponding to secret shares b_j under any key (and in particular under sk_k). Otherwise, the adversary could "remember" these ciphertexts and then adaptively choose a future leakage function applied on sk_k to decrypt it. Proceeding inductively, finally the user will compute a_{output} and since b_{output} was set initially to 0, the user has obtained $v_{output} = a_{output}$.

Finally, to prepare for another execution of $Eval(x')$ for a new x', all ciphertexts and keys containing secret shares of the bits of the secret input y are regenerated. This effectively "resets" the amount of leakage that has happened on these ciphertexts and keys. In the next execution we again (from scratch) choose a new oblivious encryption of a random b_j for each internal wire j.

Summary: One of the main advantages of the above construction was that it let us go from a procedure for blinded OC-secure homomorphic evaluation of a single (NAND) gate, and obtain an evaluation mechanism for an arbitrary functionality (using several other properties of the subsidiary cryptosystem). We note that if the subsidiary cryptosystem supports more complex homomor-

[6] Note that, in terms of leakage, this sub-computation may itself be separated into smaller sub-computations

phic computations, we may hope to use the same framework to obtain a more efficient construction, operating at the level of larger computations rather than gate-by-gate (perhaps with improved granularity). We also note that the above construction should be viewed mainly as a proof-of-concept, we did not attempt here to make it practical enough for implementation.

1.2 Related Work

Our work is inspired by many beautiful classical techniques in the field of cryptography. For one, the central idea of our compiler may be thought of as a cross between the garbled circuit method originated by Yao [Yao82] and the pioneering idea of Goldreich, Micali, and Wigderson [GMW87] of computing on data by keeping it in a secret shared form and computing on the shares. Using limited homomorphic properties of encryption schemes in order to perform reduced round oblivious circuit evaluation was proposed in the work of Sander, Young, and Yung [SYY99]. Secure hardware was proposed in many prior works in the context of achieving provable security, starting with work of Goldreich and Ostrovsky [GO96] on software protection which assumes a universal secure leak-free processor. Most importantly, our work should be compared to results in other side channel attack models. We note that in the random oracle model other works have appeared (we do not cover all these results here).

The pioneering work of Ishai, Sahai, and Wagner [ISW03] first considered the questions of converting general cryptographic algorithms (or circuits) to equivalent leakage resistant circuits. They treat leakage attacks which leak the values of an a-priori fixed number of wires of the circuit, and produce leakage resistent circuits which grow in size as a function of the total bound on number of wires which are allowed to leak. The work applies to an unbounded number of executions of the circuit, assuming leakage attacks only apply per execution. Stated differently, the assumption is that the history of all past executions is erased. This is closely inspired by the model of proactive security. In quantitative terms, they place a global bound L on the number of wires whose values leak, compile any circuit C into a new circuit of size roughly $C \cdot L^2$ which is resilient to leakage of up to L wire values (in our work the leakage bound grows with the complexity of the transformed circuit).

Faust, Tromer, Rabin, Reyzin, and Vaikuntanathan [FRR$^+$09] also address converting general cryptographic algorithms (or circuits) to equivalent leakage resistant circuits extending [ISW03] significantly. They allow a (measurement) a side channel attack on an execution to receive the result of a leakage function which takes as input the entire existing (non-erased) state of the computation (rather than values of single wires), but in return restrict the leakage functions ℓ_i that can be handled to AC^0. Quantitatively, as in [ISW03], they place a fixed bound L on the amount of leakage, and blow up the computation size by a factor of roughly L^2. [FRR$^+$09] require a secure hardware component as well.

The bounded *memory leakage model* [AGV09] has received much attention. Here one allows ℓ to be defined on the *entire contents of memory* including all stored cryptographic secret keys, all previous computation done on the secret key results, and internally generated randomness. Obviously, in this strong setting, if no erasures are incorporated in the system, one must bound *the total amount* of information that measurements can yield on the original cryptographic keys, or else they will eventually be fully leaked by the appropriate adversarial choice of ℓ. This is the model used in the works of [AGV09,NS09]. In contrast, in our work, we are interested in the continuous leakage question. Namely, the cryptographic algorithm initialized with secret cryptographic keys is invoked again and again for a (not specified in advance) polynomial (in the size of the initial cryptographic keys) number of times; each time the side-channel adversary continues to get some information on the secrets of the computation. Thus, the total amount of information that the adversary gets over the life time of the system will unbounded.

Coming back to the OC attack model, the ideas of Goldwasser, Kalai, and Rothblum [GKR08] in the work on one-time programs provide another avenue for transforming general cryptographic circuits to equivalent leakage resistant algorithms. The resulting leakage resistant algorithm will be secure in the OC attack model if it is *executed once.* To obtain an unbounded number of executions of the original circuit, one can resort to an off-line/on-line per-execution model where every execution is preceded by an off line stage in which the circuit conversion into a leakage resistent algorithm is performed a-new (obviously using new randomness). This is done prior to (and independently from) the choice of input for the coming execution. Surprisingly, the produced circuits are secure even if *all* data which is touched by the computation leaks. Namely, in presence of any polynomial time leakage functions including the identity function itself!

A recent independent work published in this proceedings is by Juma and Vahlis [JV10]. They also work in the OC attack model and address the question of how to run general computations in this model. They use as a tool a fully homomorphic encryption scheme and a leakage free hardware component independent from the functionality being computed. In terms of granularity, they divide each activation into two parts: one of which is large (a homomorphic computation of the entire circuit), and the second of which is small (a decryption). Quantitatively, To tolerate a leakage bound of L bits in total, they transform the computation into one of size $C \cdot \exp(L)$. Under stronger assumptions (e.g. sub-exponential security of the fully homomorphic encryption) the transformed computation can be of size $C \cdot \mathrm{poly}(L)$.

2 Security Definitions

2.1 Leakage Model

Leakage Attack. A leakage attack is launched on an algorithm or on a data string. In the case of a data string x, an adversary can request to see any efficiently computable function $\ell(x)$ whose output length is bounded by λ bits. In the case of an algorithm, we divide the algorithm into disjoint sub-computations. We assume that *only computation leaks information*, and so the adversary can request to see a bounded-length function of each sub-computation's input (separately).

Definition 1 (Leakage Attack $\mathcal{A}[\lambda : s](x)$). *Let s be a source: either a* data string *or a* computation. *We model a λ-bit leakage attack of adversary $\mathcal{A}$ with input x on the source s as follows.*

If s is a computation (viewed as a boolean circuit with a fixed input), it is divided into m disjoint and ordered sub-computations $sub_1, \ldots, sub_m$, where the input to sub-computation sub_i should depend only on the output of earlier sub-computations. A λ-bit Leakage Attack on s is one in which $\mathcal{A}$ can adaptively choose PPTM functions $\ell_1, \ldots \ell_m$, where ℓ_i takes as input the input to sub-computation i, and has output length at most λ bits. For each ℓ_i (in order), the adversary receives the output of ℓ_i on sub-computation sub_i's input, and then chooses ℓ_{i+1}. The view of the adversary in the attack consists of the outputs to all the leakage functions.

In the case that s is a data string, we treat it as a single subcomputation. A λ-bit leakage attack of $\mathcal{A}$ on s is one in which $\mathcal{A}$ adaptively chooses λ single-bit functions of the string in its entirety.

Multi-Source Leakage Attacks. A multi-source leakage attack is one in which the adversary gets to launch concurrent leakage attacks on several sources. Each source is an algorithm or a data string. The leakages from each of the sources can be interleaved arbitrarily, but each leakage is computed as a function of a single source only.

Definition 2 (Multi-Source Leakage Attack $\mathcal{A}[\lambda : s_1, \ldots, s_k](x)$). *Let $s_1, \ldots, s_k$ be k leakage sources (algorithms or data strings, as in Definition 1). We model a λ-bit multi-source leakage attack on $[s_1, \ldots, s_k]$ as follows. The adversary $\mathcal{A}$ with input x runs concurrently k separate λ-bit leakage attacks, one attack on each source. The attacks can be interleaved arbitrarily and adaptively. The attacks on each of the sources separately form a λ-bit leakage attack as in Definition 1. It is important that each leakage function is computed as a function of a single sub-computation in a single source (i.e. the leakages are never a function of the internal state of multiple sources). It is also important that the attacks launched by the adversary are concurrent and adaptive, and their interleaving is controlled by the adversary.*

***Simulated** Multi-Source Leakage Attacks.* For security definitions, we will occasionally want to replace the adversary's access to one or more source in a multi-source leakage attack with a view generated by a *simulator*. To facilitate composition, we view some sources as fixed: these are outside of the simulator's control. Both the adversary and the simulator get leakage access to these fixed sources (these are analogous to the environment in the UC framework [Can01]). Access to all of the other sources is simulated by the simulator.

Definition 3 (Simulated Multi-Source Leakage Attack). *Let $s_1, \ldots, s_k$ each be either a special symbol $\perp$ or a leakage source (algorithm or data string, as in Definition 1). Denote by $s'_1, \ldots, s'_\ell$ the subset of $s_1, \ldots, s_k$ that are not $\perp$. A* simulated *λ-bit multi-source leakage attack $(\mathcal{A}[\lambda : s_1, \ldots, s_k](x), \mathcal{S}[\lambda' : s'_1, \ldots, s'_\ell](x'))$ on $[s_1, \ldots, s_k]$ is defined as follows.*

$\mathcal{A}$ with input x runs concurrently k separate λ-bit leakage attacks, one attack on each of its k sources, as in Definition 2. The difference here is that the sources which are $\perp$ are all under the control of the simulator *$\mathcal{S}$. The simulator $\mathcal{S}$, which itself has an input x' and can launch a λ'-bit multi-source leakage attack on $[s'_1, \ldots, s'_\ell]$, produces the answers to all of the adversary's queries to* all of the sources *(including the $\perp$ sources).*

As in Definition 2, the adversary's (and the simulator's) access to its sources can be interleaved arbitrarily. The only difference is that the adversary's leakage queries to some of the sources are answered by the simulator. The simulator's answers may also be adaptive and depend on its prior view, which includes all of the adversary's past queries to simulated sources.

As discussed above, the motivation for including sources that are outside the simulator's control is to facilitate composition between different components that are each (on their own) resilient to multi-source leakage attacks. Throughout this work, it will be the case that $\lambda' \geq \lambda$, and so it is "easy" for the simulator to answer $\mathcal{A}$'s queries to the "non-$\perp$ sources" (by making the same query itself). The challenge is answering $\mathcal{A}$'s queries to the "$\perp$-sources".

2.2 Continuous Side-Channel Secure Compiler

We divide a side-channel-secure compiler into two parts: the first part, the *initialization* occurs only once at the beginning of time. This procedure depends only on the circuit C being compiled and the private input y. We assume that during this phase there are no side-channels. The second part is the *evaluation*. This occurs whenever the user wants to evaluate the circuit $C(\cdot, y)$ on an input x. In this part the user specifies an input x, the corresponding output $C(x, y)$ is computed, and side-channels are in effect.

Definition 4 ($\lambda(\cdot)$-Continuous Side-Channel Secure Compiler). *for a circuit family $\{C_n(x, y)\}_{n\in\mathbb{N}}$, where C_n operates on two n-bit inputs, we will say that a compiler $(Init_C, Eval_C)$ offers* $\lambda(\cdot)$-security under continuous side-channels*, if for every integer $n > 0$, every $y \in \{0,1\}^n$, and every security parameter κ, the following holds:*

- *Initialization: $Init_C(1^\kappa, C_n, y)$ runs in time $\mathrm{poly}(\kappa, n)$ and outputs an initial state $state_0$*
- *Evaluation: for every integer $t \leq \mathrm{poly}(\kappa)$, the evaluation procedure is run on the previous state $state_{t-1}$ and an input $x_t \in \{0,1\}^n$. We require that for every $x_t \in \{0,1\}^n$, when we run: $(output_t, state_t) \leftarrow Eval_C(state_{t-1}, x_t)$, with all but negligible probability over the coins of $Init_C$ and the t invocations of $Eval_C$, $output_t = C_n(x_t, y)$.*
- *$\lambda(\kappa)$-Continuous Leakage Security: for every PPTM (in κ) leakage-adversary $\mathcal{A}$, there exists a PPTM simulator $\mathcal{S}$ s.t. the view of $\mathcal{A}$ when adaptively choosing inputs $(x_1, x_2, \ldots x_T)$ while running a continuous leakage attack on the evaluation procedure, is indistinguishable from the view generates by $\mathcal{S}$ which only gets the inputs-output pairs $((x_1, C(x_1, y)), \ldots, (x_T, C(x_T, y)))$.*

Formally, the adversary repeatedly and adaptively, in iterations $t \leftarrow 1, \ldots, T$, chooses an input x_t and launches a $\lambda(\kappa)$-bit leakage attack on $Eval_C(state_{t-1}, x_t)$ (see Definition 1). The view *$view_{\mathcal{A},t}$ of the adversary in iteration t includes the input x_t, the output $output_t$, and the leakages. The complete view of the adversary is $view_{\mathcal{A}} = (view_{\mathcal{A},1}, \ldots, view_{\mathcal{A},T})$, a random variable over the coins of the adversary, of the $Init_C$ and of the $Eval_C$ procedure (in all of its iterations).*

We note that modeling the leakage attacks requires dividing the $Eval_C$ procedure into sub-computations. In our constructions the size of these sub-computations will always be at most polynomial in the security parameter.

The simulator's view is generated by running the adversary with simulated *leakage attacks. In each iteration t the simulator gets the input x_t chosen by the adversary and the circuit output $C(x_t, y)$. It generates simulated side-channel information as in Definition 3. It is important that the simulator sees nothing of the internal workings of the evaluation procedure. We compute:*

$$state_{\mathcal{S},0} \leftarrow \mathcal{S}(1^\kappa, C_n), x_t \leftarrow \mathcal{A}(view_{\mathcal{S},1}, \ldots, view_{\mathcal{S},t-1}),$$

$$(state_{\mathcal{S},0}, view_{t,\mathcal{S}}) \leftarrow \mathcal{S}(1^\kappa, x_t, C(x_t, y), view_{\mathcal{S},t-1})$$

where $view_{\mathcal{S},t}$ is a random variable over the coins of the adversary when choosing the next input and of the simulator. The complete view of the simulator is $view_{\mathcal{S}} = (view_{\mathcal{S},1}, \ldots, view_{\mathcal{S},T})$.

We require that $view_{\mathcal{S}}$ and $view_{\mathcal{A}}$ are computationally indistinguishable.

3 Subsidiary Cryptosystem and Hardware

We now present the subsidiary cryptosystem and hardware device we will use to instantiate our main construction. We also define the properties we need from the subsidiary cryptosystem. We omit the full formal details of the instantiations of these properties by the subsidiary cryptosystem for lack of space, but direct the reader back to Section 1.1 for an overview of these properties and how they are instantiated.

3.1 The Naor-Segev/BHHO Scheme and Secure Hardware

Security is based on the Decisional Diffie-Hellman (DDH) Assumption: Let *Gen* be a probabilistic group generator, s.t. $G \leftarrow Gen(1^\kappa)$ is a group of order $q = q(\kappa)$. We will take G to be $\mathbb{GF}[\mathsf{q}]$, i.e. the field of prime order q (which also supports addition operations). The DDH assumption for *Gen* is that the ensembles below are computationally indistinguishable:

$$(G, g_1, g_2, g_1^r, g_2^r) : G \leftarrow Gen(1^\kappa), g_1, g_2 \in_R G, r \in_R \mathbb{GF}[\mathsf{q}]$$

$$(G, g_1, g_2, g_1^{r_1}, g_2^{r_2}) : G \leftarrow Gen(1^\kappa), g_1, g_2 \in_R G, r_1, r_2 \in_R \mathbb{GF}[\mathsf{q}]$$

The cryptosystem has the following algorithms (we take $m = 10$, this choice is arbitrary and effects the constant in the fraction of leakage we can tolerate):

• $KeyGen(1^\kappa)$: choose $\boldsymbol{g} = (g_1, \ldots, g_m) \in_R G^m$ and $\boldsymbol{s} = (s_1, \ldots, s_m) \in_R \mathbb{GF}[\mathsf{q}]^m$. Define: $y = \prod_{i=1}^m g_i^{s_i}$. Output $pk = (\boldsymbol{g}, y)$ and $sk = \boldsymbol{s}$.
• $Encrypt(pk, b \in \{0,1\})$: parse $pk = (\boldsymbol{g}, y)$ and choose $r \in_R \mathbb{GF}[\mathsf{q}]$. Output: $c \leftarrow (g_1^r, \ldots, g_m^r, y^r \cdot g_1^b)$
• $Decrypt(sk, c)$: parse $sk = \boldsymbol{s}$ and $c = (f_1, \ldots, f_m, h)$. Compute $h' = \prod_{i=1}^m f_i^{s_i}$. Output 1 if $h = g_1 \cdot h'$ and output $\perp$ otherwise.

CipherGen *Secure Hardware.* This device will be used to realize additional useful properties for the subsidiary cryptosystem. We assume that when this device is invoked, there is leakage on its input and output, but not on its internal workings or randomness. The device receives as input a public key and mode of operation $m \in \{0, rand\}$. In mode $m = 0$ it computes and outputs a fresh encryption of 0, and in mode $m = rand$ it chooses a uniformly random bit $b \in \{0,1\}$ and outputs a fresh encryption of b.

3.2 Homomorphic and Leakage-Resilient Properties

Definition 5 (Semantic Security Under $\lambda(\cdot)$-Multi-Source Leakage). *An encryption scheme $(KeyGen, Encrypt, Decrypt)$ is semantically secure under multi-source leakage attacks if for every PPTM adversary $\mathcal{A}$, when we run the game below, the adversary's advantage in winning (over 1/2) is negligible:*

1. *The game chooses a key pair $(pk, sk) \leftarrow KeyGen(1^\kappa)$, chooses uniformly at random a bit $b \in_R \{0,1\}$, and generates a ciphertext $c \leftarrow Encrypt(pk, b)$.*
2. *The adversary launches a multi-source leakage attack on sk and c, and outputs a guess b' for the value of b:*

$$b' \leftarrow \mathcal{A}[\lambda(\kappa) : sk, c](pk)$$

The adversary wins if $b' = b$.

Lemma 1. *The Naor-Segev cryptosystem, as defined in Section 3.1, is semantically secure under $(\lambda = mq/3)$-multi-source leakage.*

Definition 6 (Key Refreshing). *An encryption scheme supports key-refreshing if it has additional algorithms with the following properties:*

1. *The key refresh procedure $Refresh(1^\kappa)$ outputs a "correlation value" π every time it is run.*
2. *The key correlation procedures output new secret and public keys $pk' \leftarrow PKCor(pk, \pi)$ and $sk' \leftarrow SKCor(sk, \pi)$. Here pk' is a public key corresponding to sk'. We require that even for fixed sk, the new sk' (as a function of a randomly chosen π) is uniformly random.*
3. *The ciphertext correlation procedure transforms an encryption from one key to the other. I.e. if $c' \leftarrow CipherCor(pk, c, \pi)$, then $Decrypt(sk, c) = Decrypt(sk', c')$.*
4. *The key linking procedure outputs a correlation value linking its two input secret keys. I.e. if $\pi \leftarrow KeyLink(sk, sk')$, then $sk' = SKCor(sk, \pi)$.*

5. *A correlation-inverter CorInvert such that* $\pi^{-1} \leftarrow \mathit{CorInvert}(\pi)$ *satisfies that if* $sk' = \mathit{SKCor}(sk, \pi)$*, then* $sk = \mathit{SKCor}(sk', \pi^{-1})$*. Also for the corresponding public keys* $pk = \mathit{PKCor}(pk', \pi^{-1})$*.*

Definition 7 ($\lambda(\cdot)$-Leakage Oblivious Generation). *An encryption scheme* $(\mathit{KeyGen}, \mathit{Encrypt}, \mathit{Decrypt})$ *supports oblivious generation if there exists a randomized procedure OblivGen such that:*

1. *OblivGen outputs the encryption of a random bit:*
$$\forall b \in \{0,1\} : \Pr_{c \leftarrow \mathit{OblivGen}(pk)}[\mathit{Decrypt}(sk, c) = b] = 1/2$$
2. *The security requirement is that there exists a Simulator* $\mathcal{S}$ *such that for every bit* $b_1 \in \{0,1\}$ *and every PPTM adversary* $\mathcal{A}$*, when we run the game below, the real and simulated views are indistinguishable:*
   (a) *The game chooses a key pair* $(pk, sk) \leftarrow \mathit{KeyGen}(1^\kappa)$*.*
   (b) *In the real view,* $\mathcal{A}$ *launches a* $\lambda(\kappa)$*-bit multi-source leakage attack:*
   $$\mathcal{A}[\lambda(\kappa) : sk, c_0 \leftarrow \mathit{OblivGen}(pk), c_0](pk)$$
   In the simulated view, the game encrypts bit b_1*:* $c_1 \leftarrow \mathit{Encrypt}(pk, b_1)$*, and we run* $\mathcal{A}$ *with a* simulated $\lambda(\kappa)$*-multi-source leakage attack:*
   $$(\mathcal{A}[\lambda(\kappa) : sk, \perp, c_1](pk), \mathcal{S}[\lambda'(\kappa) : sk, c_1](pk))$$
   I.e., here the leakage attacks on the oblivious generation procedure are simulated by $\mathcal{S}$*. We require that* $\lambda'(\kappa) = O(\lambda(\kappa))$ *(the simulator may get access to a little more leakage than the adversary).*

Definition 8 ($\lambda(\cdot)$-Leakage Ciphertext Regeneration). *An encryption scheme* $(\mathit{KeyGen}, \mathit{Encrypt}, \mathit{Decrypt})$ *supports oblivious generation if it has a procedure Regen such that:*

1. *When we run* $(pk', sk', c') \leftarrow \mathit{Regen}(pk, sk, c)$*, it is the case that* $\mathit{Decrypt}(sk', c') = \mathit{Decrypt}(sk, c)$*.*
2. *The security requirement is that for every PPTM adversary* $\mathcal{A}$ *that runs for* T *repeated regenerations, every bit* $b \in \{0,1\}$ *(determining whether the input ciphertext is an encryption of 0 or 1), the view generated by the adversary in the game below is indistinguishable.*
   (a) *The game chooses a key pair* $(pk_0, sk_0) \leftarrow \mathit{KeyGen}(1^\kappa)$ *and generates a ciphertext* $c_0 \leftarrow \mathit{Encrypt}(pk, b)$*.*
   (b) *The adversary* $\mathcal{A}$ *launches* $\lambda(\kappa)$*-bit multi-source leakage attack on* T *repeated regenerations:*
   $$\begin{aligned}\mathcal{A}[\lambda(\kappa) : \; & sk_0, c_0, (pk_1, sk_1, c_1) \leftarrow \mathit{Regen}(pk_0, sk_0, c_0), \\ & sk_1, c_1, (pk_0, c_0pk_2, sk_2, c_2) \leftarrow \mathit{Regen}(pk_1, sk_1, c_1), \\ & \ldots, \\ & sk_{T-1}, c_{T-1}, (pk_T, sk_T, c_T) \leftarrow \mathit{Regen}(pk_{T-1}, sk_{T-1}, c_{T-1})](pk_0, \ldots, pk_T)\end{aligned}$$

   We further require that the input to each sub-computation in the Regen procedure depends either on the input secret key or the input ciphertext, but never on both.

Homomorphic Masked NAND. A homomorphic masked NAND computation is given three ciphertexts c_1, c_2, c_3 encrypted under the same key and with corresponding plaintexts $b_1, b_2, b_3 \in \{0,1\}$, and two plain-text values $a_1, a_2 \in \{0,1\}$. It should compute homomorphically (without using the secret key) compute a "blinded" (see below) ciphertext hc that can later be decrypted to retrieve the value $((a_1 \oplus b_1)$ NAND $(a_2 \oplus b_2)) \oplus b_3$

Definition 9 ($\lambda(\cdot)$-Leakage Blinded Homomorphic NAND). *An encryption scheme (KeyGen, Encrypt, Decrypt) supports blinded homomorphic masked NANDs if there exist procedures HomEval and HomDecrypt such that:*

1. *When take $hc \leftarrow HomEval(pk, a_1, a_2, c_1, c_2, c_3)$, for the secret key sk corresponding to pk w.h.p. it holds that $HomDecrypt(sk, hc) = ((a_1 \oplus b_1)$ NAND $(a_2 \oplus b_2)) \oplus b_3$.*
2. *The result should be "blinded". There exists a Simulator $\mathcal{S}$ such for every PPTM adversary $\mathcal{A}$, PPTM ciphertext generators G_1, G_2, G_3,[7] and plaintext values $a_1, a_2 \in \{0,1\}$, the real and simulated views in the game below are indistinguishable:*
   (a) *The game chooses a key pair $(pk, sk) \leftarrow KeyGen(1^\kappa)$ and generates ciphertexts $c_1 \leftarrow G_1(pk), c_2 \leftarrow G_2(pk), c_3 \leftarrow G_3(pk)$ using random strings r_1, r_2, r_3 for G_1, G_2, G_3 respectively.*
   (b) *In the real view, the adversary $\mathcal{A}$ launches a multi-source leakage attack on the homomorphic evaluation and decryption:*

$$\begin{aligned}\mathcal{A}[\lambda(\kappa) : sk, c_3 &\leftarrow G_3(r_3),\\ hc &\leftarrow HomEval(pk, a_1, a_2, c_1, c_2, c_3),\\ a_3 &\leftarrow HomDecrypt(sk, hc)](pk, a_1, a_2, r_1, r_2)\end{aligned}$$

In the simulated view, the simulator does not get any access to homomorphic evaluation or decryption, but rather gets only the output a_3 of the homomorphic decryption:

$$\begin{aligned}(\ \ &\mathcal{A}[\lambda(\kappa) : sk, c_3 \leftarrow G_3(r_3), \bot, \bot](pk, a_1, a_2, r_1, r_2),\\ &\mathcal{S}[\lambda'(\kappa) : sk, c_3 \leftarrow G_3(r_3)](pk, a_1, a_2, r_1, r_2, a_3))\end{aligned}$$

We require that $\lambda'(\kappa) = O(\lambda(\kappa))$.

4 A Continuous-Leakage Secure and Compiler

The compiler can be based on any subsidiary cryptosystem with the properties of Section 3. We refer the reader to Section 1.1 for the construction overview and preliminaries, and to Section 2.2 for the security definition. The initialization and evaluation procedures are presented below in Figure 1. The evaluation procedure

[7] In the security proof for our construction these generation procedures will be the *OblivGen* or *Regen* procedure.

is separated into sub-computations (which may themselves be separated into sub-computations of the cryptographic algorithms). For each such sub-computation we explicitly note which data elements are computed on ("touched") in the sub-computation. We defer the proof of security to the full version.

Initialization $\mathit{Init}_C(1^\kappa, C, y)$

For every input wire i, corresponding to bit j of the input y, generate new keys: $(pk_i, sk_i) \leftarrow \mathit{KeyGen}(1^\kappa)$ and compute an encryption $c_i = \mathit{Encrypt}(pk_i, y_j)$. $\mathit{state}_0 \leftarrow \{(pk_i, sk_i, c_i)\}_{i\,:\,i \text{ is a } y\text{-input wire}}$

Evaluation $\mathit{Eval}_C(\mathit{state}_{t-1}, x_t)$

1. *Generate keys and ciphertexts for all wires of C except the y-input wires.*
   For the x input wires, generate fresh keys and encryptions of 0.
   Proceed layer-by-layer (from input to output). For each gate g with input wires i and j and output wire k: (repeat independently for gate g's second output wire ℓ)
   (a) Generate a random correlation value $\pi_{i,k} \leftarrow \mathit{Refresh}(1^\kappa)$. Apply this value to wire i's keys to get a new key pair for wire k: $pk_k \leftarrow \mathit{PKCor}(pk_i, \pi_{i,k})$, $sk_k \leftarrow \mathit{SKCor}(sk_i, \pi_{i,k})$. Derive a correlation value specifying the correlation between the keys of wires j and k: $\pi_{j,k} \leftarrow \mathit{KeyLink}(sk_k, sk_j)$. Store the keys and correlation values. *"Computed on" keys, correlation values*
   (b) Generate a ciphertext encrypting the share b_k for wire k: for internal wires, use the oblivious generation procedure to generate an encryption of a random bit $c_k \leftarrow \mathit{OblivGen}(pk_k)$.
   For the output wire o, generate an encryption $c_o \leftarrow \mathit{Encrypt}(pk_o, 0)$.
   Store the ciphertexts. *"Computed on" ciphertexts*
2. *Compute the value of $C(y, x_t)$.*
   Proceed layer by layer (from input to output). For each gate g with output wire k and input wires i, j, the previous gate evaluations yield the shares $a_i, a_j \in \{0, 1\}$ of the gate's input wires. Compute an encryption of a_k: (do the same independently for gate g's second output wire ℓ):
   (a) First transform the ciphertexts c_i and c_j to be encryptions under pk_k: $c'_i \leftarrow \mathit{CipherCor}(pk_i, c_i, \pi_{i,k})$ and $c'_j \leftarrow \mathit{CipherCor}(pk_j, c_j, \pi_{j,k})$. *"Computed on" ciphertexts and correlation values.*
   (b) Run the blinded homomorphic evaluation procedure: $hc_k \leftarrow \mathit{HomEval}(pk_k, a_i, a_j, c'_i, c'_j, c_k)$. *"Computed on" ciphertexts.*
   (c) Compute $a_k \leftarrow \mathit{HomDecrypt}(sk_k, hc_k)$. *"Computed on" hc_k and the secret key.*
   Taking o to be the output wire, the output is $\mathit{output}_t \leftarrow a_o$.
3. *Generate the new state.*
   For each y-input wire i regenerate wire i's keys and ciphertext: $(pk_i, sk_i, c_i) \leftarrow \mathit{Regen}(pk_i, sk_i, c_i)$.
   The new state is $\mathit{state}_t \leftarrow \{(i, pk_i, sk_i, c_i)\}_{i\,:\,i \text{ is a } y\text{-input wire}}$.

Fig. 1. Init_C, performed off-line without side channels, and Eval_C, performed on input x_t in the presence of side-channel attacks.

Theorem 1. *Let $(\mathit{KeyGen}, \mathit{Encrypt}, \mathit{Decrypt})$ be a subsidiary encryption scheme with security parameter κ and with the properties specified in Definitions 6 (key refreshing), 5 (multi-source leakage resilience), 7 (oblivious generation), 8 (leakage resilience regeneration), and 9 (homomorphic masked NAND), all* *with $\lambda = \Omega(\kappa)$-leakage resilience.*[8] *Then the $(Init_C, Eval_C)$ compiler specified in Figure 1 offers $\Omega(\kappa)$-leakage security under continuous side-channels as in Definition 4.*

References

[AGV09] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *TCC*, pages 474–495, 2009.

[BHHO08] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky. Circular-secure encryption from decision diffie-hellman. In *CRYPTO*, pages 108–125, 2008.

[BKKV10] Zvika Brakerski, Yael Tauman Kalai, Jonathan Katz, and Vinod Vaikuntanathan. Cryptography resilient to continual memory leakage. Cryptology ePrint Archive, Report 2010/278, 2010.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CG88] Benny Chor and Oded Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM J. Comput.*, 17(2):230–261, 1988.

[DHLAW10] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana Lopez-Alt, and Daniel Wichs. Efficient public-key cryptography in the presence of key leakage. Cryptology ePrint Archive, Report 2010/154, 2010.

[DP08] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. *Annual IEEE Symposium on Foundations of Computer Science*, pages 293–302, 2008.

[FKPR09] Sebastian Faust, Eike Kiltz, Krzysztof Pietrzak, and Guy Rothblum. Leakage-resilient signatures. Cryptology ePrint Archive, Report 2009/282, 2009. http://eprint.iacr.org/2009/282.

[FRR$^+$09] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting against computationally bounded and noisy leakage. Manuscript, 2009.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *CRYPTO*, pages 39–56, 2008.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.

[JV10] Ali Juma and Yevgeniy Vahlis. On protecting cryptographic keys against continual leakage. Cryptology ePrint Archive, Report 2010/205, 2010.

[8] We mean that there exists an explicit constant $0 < c < 1$ s.t. we allow leakage of $c \cdot \lambda$ bits

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Wiener, editor, *CRYPTO99*, pages 388–397, 1999.

[KV09] Jonathan Katz and Vinod Vaikuntanathan. Signature schemes with bounded leakage resilience. In *ASIACRYPT*, pages 703–720, 2009.

[MR04] Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In *TCC*, pages 278–296, 2004.

[NS09] Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. In *CRYPTO*, pages 18–35, 2009.

[Pie09] Krzysztof Pietrzak. A leakage-resilient mode of operation. In *EUROCRYPT*, pages 462–482, 2009.

[RCL] Boston University Reliable Computing Laboratory. Side channel attacks database. http://www.sidechannelattacks.com.

[SYY99] T. Sander, A. Young, and M. Yung. Non-interactive cryptocomputing for nc^1. In *FOCS*, 1999.

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. *Symposium on Foundations of Computer Science*, 0:80–91, 1982.