Leakage-Resilient Circuits without Computational Assumptions[⋆]

Stefan Dziembowski[1,⋆⋆] and Sebastian Faust[2,⋆⋆⋆]

[1] University of Warsaw and Sapienza University of Rome
[2] Aarhus University

Abstract. Physical cryptographic devices inadvertently leak information through numerous side-channels. Such leakage is exploited by so-called side-channel attacks, which often allow for a complete security breache. A recent trend in cryptography is to propose formal models to incorporate leakage into the model and to construct schemes that are provably secure within them.

We design a *general* compiler that transforms *any* cryptographic scheme, e.g., a block-cipher, into a functionally equivalent scheme which is resilient to any *continual* leakage provided that the following three requirements are satisfied: (i) in each observation the leakage is bounded, (ii) different parts of the computation leak independently, and (iii) the randomness that is used for certain operations comes from a simple (non-uniform) distribution. In contrast to earlier work on leakage resilient circuit compilers, which relied on computational assumptions, our results are purely *information-theoretic*. In particular, we do not make use of public key encryption, which was required in all previous works.

1 Introduction

Leakage resilient cryptography attempts to incorporate side-channel information leakage into standard cryptographic models and to design new cryptographic schemes that provably withstand such leakages under reasonable physical assumptions. The "holy grail" in leakage-resilient cryptography is a *generic* method to provably protect *any* cryptographic computation against a broad, well-defined and realistic class of side-channel leakages. This fundamental question has first

[⋆] This work was supported by the *WELCOME/2010-4/2* grant founded within the framework of the *EU Innovative Economy (National Cohesion Strategy) Operational Programme.*

[⋆⋆] The European Research Council has provided financial support to the first author of this paper under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no CNTM-207908.

[⋆⋆⋆] Sebastian Faust acknowledges support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, within part of this work was performed; and from the CFEM research center, supported by the Danish Strategic Research Council.

been studied in the work of Ishai et al. [ISW03] who initiated the concept of *leakage resilient circuit compilers.* A circuit compiler takes a description of a (Boolean) circuit Γ as input and outputs a transformed (Boolean) circuit Π^{Γ} with the same functionality, but with resilience to certain well-defined classes of leakage. The authors consider a very specific type of leakage, namely, an adversary who learns the values of up to $n \in \mathbb{N}$ internal wires in each execution of Π^{Γ}. Security is proven by a simulation based argument. More precisely, it is shown that any (computationally unbounded) adversary that learns the value of up to n internal wires in each execution of Π^{Γ} has only a negligible advantage over an adversary that only views the inputs/outputs of the original circuit Γ.

The result of Ishai et al. shows security for a very restricted class of leakages, namely, security is proven only against the *specific attack* of learning the values of n wires. The question that motivates our work is whether, analogously to [ISW03], we can protect any computation against the much broader class of *polynomial-time computable leakages.* This question has been answered affirmatively in the recent feasibility results of Juma and Vahlis [JV10] and Goldwasser and Rothblum [GR10] by making additionally use of the prominent "only computation leaks information" assumption [MR04]. The security of both compilers, however, relies on heavy cryptographic machinery by using public key encryption to "encrypt" the secret state and the whole computation of Γ.[1]

At first sight, it may look natural to rely on some form of cryptographic encryption, if we want to achieve security against any polynomial-time computable leakage function. For instance, it is necessary to "encrypt" the secret state of Γ, as already a single bit of information leaking about the original secret state makes simulation-based security impossible. Perhaps surprisingly, in this paper we show that cryptographically secure encryption schemes are not necessary to construct leakage resilient circuit compilers for *polynomial-time computable leakages.* More precisely, we show that even an *unbounded adversary* with continuous leakage access to Π^{Γ} only gains a negligible advantage over an adversary with only black-box access to Γ.

Similar to earlier work, we make certain restrictions on the leakage. We follow the work of Dziembowski and Pietrzak [DP08], and allow the leakage to be arbitrary as long as the following two restrictions are satisfied:

1. **Bounded leakage:** the amount of leakage in each round is bounded to λ bits (but overall can be arbitrary large).
2. **Independent leakage:** the computation can be structured into sub computations, where each part of the computations leaks independently (we define the term of a "sub computation" below).

Formally, this is modeled by letting the adversary for each observation choose a leakage function f with range $\{0,1\}^{\lambda}$, and then giving her $f(\tau)$ where τ is all the data that has been accessed in the current sub-computation. In addition, we require access to a source of correlated randomness generated in a *leak-free* way

[1] More precisely, Juma and Vahlis require fully homomorphic encryption, while Goldwasser and Rothblum use a variant of the BHHO encryption scheme.

– e.g., computed by a simple leak free component. We provide more details on our hardware assumptions below.

On Independent Leakages. Variants of the assumption that different parts of the computation leak independently have been used in several works [DP08, Pie09, KP10, GR10, GR10, JV10]. In its weakest form, the assumption says that the state is divided into two parts that leak independently. This type of assumption is used, e.g., in the work on leakage resilient stream ciphers [DP08, Pie09]. Several stronger flavors have been used in the literature. For instance, in the circuit compiler of Goldwasser and Rothblum [GR10] the computation is structured into $O(s)$ sub-computations, where s is the size of the original circuit. Of course, in practice leakage is a global phenomenon and assumptions that require a large number of independent computations is a strong assumption on the hardware. We would like to emphasize, however, hat many relevant global leakage functions can be computed from independent leakages. This is not only true for the prominent Hamming weight leakage, but more generally, for *any affine* leakage function.

On the Relation between Leakage Granularity and the Amount of Leakage. We show a relation between the granularity level of the independent leakage assumption and the amount of leakage that can be tolerated per observation. More precisely, in our basic setting we assume that the computation is structured into $2s$ parts that leak independently, where s is the number of gates in Γ (this is comparable to the model of [GR10]). Here, the amount of leakage can increase linearly with the size of the circuit. Alternatively, we may settle for weaker independency assumptions. That is, in the best case we may require only *two sub-components* that leak independently. Of course this comes at a price: the amount of leakage that is tolerated is *independent* of the circuit's size. We notice that we can tolerate more leakage if we assume some strong form of *memory erasures* between sub-computations (cf. Section 6 for the details).

On Leak-Free Components. Leak-free components are used by recent leakage resilient circuit compilers [GKR08, FRR$^+$10, JV10, GR10]. A leak-free component leaks from its outputs, but the leakage is oblivious to its internals. In this work, we use the leak-free component, $\mathcal{O}$, that was recently introduced by Dziembowski and Faust [DF11]. This component outputs two random vectors $A, B \leftarrow \mathbb{F}^n$ (with $\mathbb{F}$ being a finite field and n being a statistical security parameter) such that their inner product is 0, i.e., $\sum_i A_i \cdot B_i = 0$. As discussed in [DF11], $\mathcal{O}$ exhibits several properties that are beneficial for implementation. We refer the reader to [DF11] for a more thorough discussion on the properties of $\mathcal{O}$.

1.1 Our Contributions

We propose a general transformation (also called the "compiler") that takes any circuit Γ computing over finite fields $\mathbb{F}$ and transforms it into Π^Γ in such a way that (1) the circuit Π^Γ computes the same function as Γ, and (2) any (computationally unbounded) adversary that obtains continuous leakage from

Π^Γ gains only negligible advantage over an adversary with only black-box access to Γ. We emphasize that in contrast to earlier works in similar leakage models [GR10, JV10], we do not use public key encryption to achieve leakage resilience. This makes our results significantly more efficient.

Our construction is secure in the continuous leakage setting with adaptive queries. That is, we assume that the circuit Γ can be initialized (during a trusted step-up phase) with some secret *state*, and is then queried by an adversary $\mathcal{S}$ on adaptively chosen inputs $X^1, \ldots, X^\ell$. For each i let $Y^i := \Gamma(X^i, \mathit{state})$ be the outcome of the ith query. To define security, we consider an adversary $\mathcal{A}$ that attacks Π^Γ and gets the same information (i.e., pairs $(X^1, Y^1), \ldots, (X^\ell, Y^\ell)$ for X^i's chosen by him) *plus* the leakage from each computation. Informally, the security definition requires that for every such (computationally unbounded) adversary $\mathcal{A}$, there exists $\mathcal{S}$ with only black-box access to Γ that produces the same output as $\mathcal{A}$. The formal definition is given in Section 5.3. For simplicity, in the formal model we consider only the case where the adversary is allowed to observe the computation once. For readers familiar with the work on leakage resilient circuits [ISW03,FRR$^+$10] this is the case of stateless circuits. We briefly discuss how to extend our result to the continuous leakage setting in Section 6.

We emphasize that the running time of our simulator $\mathcal{S}$ is polynomial in the running time of $\mathcal{A}$. This is necessary to protect circuits Γ, which hide the secret key only computationally – which is the case for most prominent cryptographic schemes. This is in contrast to the recent work of Dziembowski and Faust [DF11] that consider efficient transformations for cryptographic schemes which hide the secret key information theoretically (e.g., Okamoto signatures or Cramer-Shoup encryption).

1.2 Comparison to Related Work

An extension of the circuit compiler of Ishai et al. [ISW03] (mentioned above) was proposed by Faust et al. [FRR$^+$10]. The authors use similar techniques as [ISW03] based on secret sharing but give a significantly improved security analysis considering computationally weak (e.g., AC0) and noisy leakages. Similar to our work, the results of [ISW03,FRR$^+$10] work in the information theoretic setting. The leak-free components that are used in earlier works are similar in spirit to the component used in our work. In [FRR$^+$10], the leak-free component outputs an n-bit string with parity 0, while in the works of Juma and Vahlis [JV10] and Goldwasser and Rothblum [GR10] it outputs ciphertexts that encrypt 0 using the underlying public-key encryption scheme. Except for the work of Juma and Vahlis all leakage resilient circuit compilers (including ours) require at least one leak-free component for each gate in the original circuit Γ.

We finally remark that our results do not imply the recent results of Dziembowski and Faust [DF11]. More precisely, although we use the same trusted source $\mathcal{O}$ as [DF11], the schemes of [DF11] cannot be obtained by using our circuit compiler. The reason for this are twofold: first, the protocols of [DF11] only use the leak-free component *for the refreshing* of the secret key, while our protocols need to use $\mathcal{O}$ for each gate of the original circuit. Second, their implementation of standard cryptographic schemes are significantly more efficient: while we work on the gate level and blow-up the circuit's size by $O(n^4)$, Dziembowski and Faust directly exploit homomorphic properties of cryptographic schemes and increase the size only by a factor of $O(n)$. Unfortunately, however, these techniques are limited only to certain schemes such as the Okamoto identification and the Cramer-Shoup encryption.

2 Preliminaries

For a set $\mathcal{S}$ we denote by $X \leftarrow \mathcal{S}$ the process of drawing X uniformly from $\mathcal{S}$. A vector V is a row vector, and we denote by V^{T} its transposition. We let $\mathbb{F}$ be a finite field and for $m, n \in \mathbb{N}$, let $\mathbb{F}^{m\times n}$ denote the set of $m \times n$-matrices over $\mathbb{F}$. For a matrix $M \in \mathbb{F}^{m\times n}$ and an m bit vector $V \in \mathbb{F}^m$ we denote by $V \cdot M$ the n-element vector that results from matrix multiplication of V and M. For a natural number n let $(0)^n = (0, \dots, 0)$. We use $V[i]$ to denote the ith element of a vector V and $V[i, \dots, j]$ to denote the elements $i, i+1, ..., j$ of V. For two vectors $V \in \mathbb{F}^m, W \in \mathbb{F}^n$ we denote by $V||W$ its concatenation and by $V \otimes W$ we will mean a vector in $\mathbb{F}^{m\cdot n}$ defined as

$$V \otimes W := (V_1W_1, \dots, V_1W_m,\ V_2W_1, \dots, V_2W_m,\ \dots\ , V_nW_1, \dots, V_nW_m). \quad (1)$$

Finally, let $\langle V, W\rangle$ denote the inner product of V and W. We will use the fact that the inner product is linear, i.e. $\langle a \cdot V + V', W\rangle = a \cdot \langle V, W\rangle + \langle V', W\rangle$.

The "$\overset{d}{=}$" symbol denotes the equality of two distributions. For two random variables X_0, X_1 over $\mathcal{X}$ we define the statistical distance between X and Y as $\Delta(X; Y) = \sum_{x\in\mathcal{X}} 1/2|\Pr[X_0 = x] - \Pr[X_1 = x]|$.

2.1 Leakage Model

To formally model leakage, we follow Dziembowski and Faust [DF11] and only recall some important details here. We model independent leakage from memory parts in form of a *leakage game*, where the adversary can *adaptively* learn information from the memory parts. More precisely, for some $c, \ell, \lambda \in \mathbb{N}$ let $M_1, \dots, M_\ell \in \{0,1\}^c$ denote the contents of the memory parts, then we define a λ-*leakage game* played between an adaptive adversary $\mathcal{A}$, called a λ-*limited leakage adversary*, and a *leakage oracle* $\Omega(M_1, \dots, M_\ell)$ as follows. For some $m \in \mathbb{N}$, the adversary $\mathcal{A}$ can adaptively issue a sequence $\{(x_i, f_i)\}_{i=1}^m$ of requests to the oracle $\Omega(M_1, \dots, M_\ell)$, where $x_i \in \{1, \dots, \ell\}$ and $f_i : \{0,1\}^c \to \{0,1\}^{\lambda_i}$ with $\lambda_i \leq \lambda$. To each such a query the oracle replies with $f_i(M_{x_i})$ and we say that in this case the adversary $\mathcal{A}$ *retrieved the value* $f_i(M_{x_i})$ *from* M_{x_i}. The only restriction is that in total the adversary does not retrieve more than λ bits from each memory part. In the following, let $(\mathcal{A} \rightleftarrows (M_1, \dots, M_\ell))$ be the output of $\mathcal{A}$ at the end of this game. Without loss of generality, we assume that $(\mathcal{A} \rightleftarrows (M_1, \dots, M_\ell)) := (f_1(M_{x_1}), \dots, f_m(M_{x_m}))$.

Leakage from Computation. We model the computation that is carried out on a device as a ℓ-party protocol $\Pi = (P_1, \dots, P_\ell)$, which is executed between the parties $(P_1, \ldots, P_\ell)$ and an adversary is allowed to obtain partial information (the leakage) from the internal state of the players. Initially, some parties may hold inputs, and we denote by S_i the input of P_i. The execution of Π with initial inputs $S_1, \ldots, S_\ell$, denoted by $\Pi(S_1, \ldots, S_\ell)$, is structured into sub-computations. In each sub-computation one player is active and sends messages to the other players. These messages can depend on his input (i.e., his initial state), his local randomness, and the messages that he received in earlier rounds. At the end of the protocol's execution, the players $P_1, \ldots, P_\ell$ output values $S'_1, \ldots, S'_\ell$, resp. (some of these values may be empty). For each player P_i, we denote the local randomness that is used by P_i during the execution of Π and all the messages that are received *or* sent (including the messages from the user of the protocol) by view_i. We assume that after the protocol terminates, the adversary $\mathcal{A}$ plays a λ-leakage game against the leakage oracle $\Omega(\mathsf{view}_i, \ldots, \mathsf{view}_\ell)$. We will use the following convention in order to simplify the exposition: while describing a protocol we will explicitly describe the view of each player, sometimes omitting redundant variables. For instance, if the view contains variables X, Y, Z, such that always $Z = X \oplus Y$, then we will omit Z, as it can be calculated by the leakage function from X and Y.

2.2 Leakage-Resilient Storage

Davi et al. [DDV10] recently introduced the notion of leakage-resilient storage (LRS) $\Phi = (\mathsf{Encode}, \mathsf{Decode})$. An LRS allows to store a secret in an "encoded form" such that even given leakage from the encoding no adversary learns information about the encoded values. One of the constructions that the authors propose uses two source extractors and can be shown to be secure in the independent leakage model. More precisely, an LRS for the independent leakage model is defined for message space $\mathcal{M}$ and encoding space $\mathcal{L} \times \mathcal{R}$ as follows:

– $\mathsf{Encode} : \mathcal{M} \to \mathcal{L} \times \mathcal{R}$ is a probabilistic, efficiently computable function and
– $\mathsf{Decode} : \mathcal{L} \times \mathcal{R} \to \mathcal{M}$ is a deterministic, efficiently computable function such that for every $S \in \mathcal{M}$ we have $\mathsf{Decode}(\mathsf{Encode}(S)) = S$.

An LRS Φ is said to be (λ, ϵ)-secure, if for any $S, S' \in \mathcal{M}$ and any λ-limited adversary $\mathcal{A}$, we have $\Delta(\mathcal{A} \rightleftarrows (L, R); \mathcal{A} \rightleftarrows (L', R')) \leq \epsilon$, where $(L, R) \leftarrow \mathsf{Encode}(S)$ and $(L', R') \leftarrow \mathsf{Encode}(S')$, for any two secrets $S, S' \in \mathcal{M}$. In this paper, we consider a leakage-resilient storage scheme $\Phi^n_{\mathbb{F}}$ that allows to efficiently store elements from $\mathcal{M} = \mathbb{F}$. It is a variant of a scheme proposed in [DF11] and based on the inner-product extractor. For some security parameter $n \in \mathbb{N}$, $\Phi^n_{\mathbb{F}} := (\mathsf{Encode}^n_{\mathbb{F}}, \mathsf{Decode}^n_{\mathbb{F}})$ is defined as follows:

– $\mathsf{Encode}^n_{\mathbb{F}}(S)$:
  1. Sample $(L[2, \ldots, n], R[2, \ldots, n]) \leftarrow \left(\mathbb{F}^{n-1}\right)^2$.
  2. Set $L[1] \leftarrow \mathbb{F} \setminus \{0\}$ and $R[1] := L[1]^{-1} \cdot (S - \langle (L[2, \ldots, n], R[2, \ldots, n]) \rangle)$

  Output (L, R).
– $\mathsf{Decode}^n_{\mathbb{F}}(L, R)$: Output $\langle L, R \rangle$.

The property that $L[1] \neq 0$ will be useful in the "generalized multiplication" protocol (cf. Section 4.2). It is easy to see that $\Phi_{\mathbb{F}}^n$ is correct, i.e.:

$$\mathsf{Decode}_{\mathbb{F}}^n(\mathsf{Encode}_{\mathbb{F}}^n(S)) = S.$$

Security is shown in the following lemma whose proof appears in the full version of this paper.

Lemma 1. *Let $n \in \mathbb{N}$ and let $\mathbb{F}$ such that $|\mathbb{F}| = \Omega(n)$. For any $1/2 > \delta > 0, \gamma > 0$ the LRS $\Phi_{\mathbb{F}}^n$ as defined above is (λ, ϵ)-secure, with $\lambda = (1/2 - \delta)n \log |\mathbb{F}| - \log \gamma^{-1} - 1$ and $\epsilon = 2m(|\mathbb{F}|^{3/2 - n\delta} + |\mathbb{F}|\, \gamma)$.*

We instantiate Lemma 1 with concrete parameters in the next corollary.

Corollary 1. *Suppose $|\mathbb{F}| = \Omega(n)$. Then, LRS $\Phi_{\mathbb{F}}^n$ is $(0.49 \cdot \log_2 |\mathbb{F}^n| - 1, negl(n))$-secure, for some negligible function negl.*

3 An Informal Description of the Protocol

In this section we describe informally our circuit compiler that is based on the LRS scheme $\Phi_{\mathbb{F}}^n$. Our starting point is the result of [DF11] where a protocol $\mathsf{Refresh}_{\mathbb{F}}^n$ is proposed to refresh secrets encoded with $\Phi_{\mathbb{F}}^n$. $\mathsf{Refresh}_{\mathbb{F}}^n$ is run between two parties P_{L} and P_{R}, which initially hold L and R in $\mathbb{F}^n$. At the end of the protocol, P_{L} holds L' and P_{R} holds R' such that $\langle L, R \rangle = \langle L', R' \rangle$. The protocol can be repeated continuously to refresh the encoding and satisfies the following security requirement: even given continuous leakage independently from the parties P_{L} and P_{R} no adversary can learn the encoded secret $\langle L, R \rangle$.

In order to create a general circuit compiler in the independent leakage model, all we need is to perform in a leakage-resilient way arithmetic operations on the encoded secrets using the LRS $\Phi_{\mathbb{F}}^n$. This is similar to the methods used in the MPC literature: first, the secret is secret-shared between the parties (in our case: "encoded"), and then the operations are performed "gate-by-gate" in a secure way. At the end the outputs of the computation are reconstructed in the following way: one of the players, P_{L}, say, sends his share L' of the output to P_{R} and P_{R} computes $\mathsf{Decode}_{\mathbb{F}}^n(L, R)$. We us a similar approach in this paper.

To illustrate this approach, consider the simple case of a circuit that multiplies a constant α with a secret S encoded as (L, R). If L is held by P_{L} and R is held by P_{R}, then one of the players, P_{L}, say, multiplies his vector by α (as $\langle \alpha \cdot L, R \rangle = \alpha \cdot \langle L, R \rangle$). Also, addition of a constant c to S is simple: the player P_{L} sends $x = L[1]$ to P_{R} (for simplicity assume that $L[1] \neq 0$), and then P_{R} sets $R' = R + (x^{-1} \cdot c, 0, \ldots, 0)$ and P_{L} sets $L' = L$. We notice that (L', R') was computed from (L, R) just by sending one field element from P_{L} to P_{R}, and in particular it did not involve computing $\langle L, R \rangle$. We call this protocol $\mathsf{AddConst}_{\mathbb{F}}^n(\alpha, (L, R))$.

The only ingredient that is missing for computing arbitrary functionalities is a protocol for leakage-resilient multiplication of two encoded secrets. The construction of such a protocol is the main contribution of this paper (for technical reasons, we construct in Section 4.2 a protocol for a slightly more general functionality, which we call "generalized multiplication"). Suppose we have two secrets $S^0 \in \mathbb{F}$ and $S^1 \in \mathbb{F}$ encoded as (L^0, R^0) and (L^1, R^1), respectively. Suppose further that player P_{L} holds (L^0, L^1) and player P_{R} holds (R^0, R^1). Their goal is to compute $L'', R'' \in \mathbb{F}^n$ in a leakage-resilient way such that $\langle L'', R'' \rangle = S$ and L'' is held by P_{L}, while R'' is held by P_{R}. Our first observation is that $\langle L^0 \otimes L^1, R^0 \otimes R^1 \rangle = \langle L^0, R^0 \rangle \cdot \langle L^1, R^1 \rangle = S^0 \cdot S^1$, which follows from simple linear algebra. Hence, $(L^0 \otimes L^1, R^0 \otimes R^1)$ encodes the secret $S^0 \cdot S^1$ in the $\Phi_{\mathbb{F}}^{n^2}$ scheme. Note that this protocol, so far, is non-interactive so it is clearly secure. The disadvantage of this protocol is that the length of the encoding grows exponentially with the depth of Γ. Therefore, we need a method of reducing the length of this encoding. This can be done in the following way. First, the players refresh the $(L^0 \otimes L^1, R^0 \otimes R^1)$ encoding with the $\mathsf{Refresh}_{\mathbb{F}}^{n^2}$ protocol. Let $(L', R') \in \mathbb{F}^{n^2} \times \mathbb{F}^{n^2}$ be the result of this refreshing. Then, the players reconstruct *in clear* the secret encoded by the final $n(n-1)$ elements of L' and R'. More precisely, the player P_{L} sends $L'[n+1, \ldots, n^2]$ to P_{R}, and P_{R} computes $d = \langle L'[n+1, \ldots, n^2], R'[n+1, \ldots, n^2] \rangle$. We now clearly have that $S^0 \cdot S^1 = \langle L', R' \rangle = \langle L'[1, \ldots, n], R'[1, \ldots, n] \rangle + d$. Hence, $(L'[1, \ldots, n], R'[1, \ldots, n])$ encodes $S^0 \cdot S^1$ minus d. Since d can be published by P_{R} we can now use the protocol $\mathsf{AddConst}_{\mathbb{F}}^n(d, (L'[1, \ldots, n], R'[1, \ldots, n]))$, and add a constant d to $(L'[1, \ldots, n], R'[1, \ldots, n])$. The output (L'', R'') of the protocol is the result of this operation. Observe that the use of the refreshing protocol is crucial, as $(L^0 \otimes L^1)[n+1, \ldots, n^2]$ gives almost complete information about L^0 and L^1.

4 The Ingredients

In this section, we describe the two main ingredients of our compiler construction: the "refreshing" protocol for $\Phi_{\mathbb{F}}^n$ (cf. Section 4.1) and the "generalized multiplication" protocol (cf. Section 4.2). The latter protocol will use the former as a sub-routine. In the full version of this paper, we show that these two components satisfy a simple security property called reconstructibility. This notion was introduced recently in [FRR+10] and essentially says that the view of the parties in a protocol can be efficiently reconstructed from just knowing the encoded inputs and outputs. For our setting, we modify this notion and define reconstruction as a protocol run between players P_{L} and P_{R}, where the efficiency criteria of the reconstructor is the amount of information exchanged between the parties. For instance, for the generalized multiplication the reconstructor protocol is run between P_{L} with input (L^0, L^1, L'') and P_{R} with input (R^0, R^1, R'') and computes view_L and view_R with only one field element of communication.

4.1 Leakage-Resilient Refreshing of LRS

In this section, we propose a simple variant of the refreshing protocol proposed in [DF11] (cf. Section 3) for the LRS $\Phi_{\mathbb{F}}^n$. As described in the introduction, we assume that the players have access to a leak-free component that samples uniformly at random pairs of orthogonal vectors. Technically, we will assume that we have an oracle $\mathcal{O}'$ that samples a uniformly random vector $((A,\tilde{A}),(B,\tilde{B})) \in (\mathbb{F}^n)^4$, subject to the constraint that the following three conditions hold:

1. $\langle A, B\rangle + \langle \tilde{A}, \tilde{B}\rangle = 0$,
2. $A \neq (0)^n$, and
3. $\tilde{B} \neq (0)^n$.

Note that this oracle is different from the oracle $\mathcal{O}$ described in the introduction (and used earlier in [DF11]) that simply samples pairs (A, B) of orthogonal vectors. It is easy to see, however, that this "new" oracle $\mathcal{O}'$ can be "simulated" by the players that have access to $\mathcal{O}$ that samples pairs (C, D) of orthogonal vectors of length $2n$ each. First, observe that $C \in \mathbb{F}^{2n}$ can be interpreted as a pair $(A,\tilde{A}) \in (\mathbb{F}^n)^2$ (where $A||\tilde{A} = C$), and in the same way $D \in \mathbb{F}^{2n}$ can be interpreted as a pair $(B,\tilde{B}) \in (\mathbb{F}^n)^2$ (where $B||\tilde{B} = D$). By the basic properties of the inner product we get that $\langle A, B\rangle + \langle \tilde{A}, \tilde{B}\rangle = \langle C, D\rangle = 0$. Hence, Condition 1 is satisfied. Conditions 2 and 3 can simply verified by players P_{L} and P_{R} respectively. If one these conditions is not met, then the players sample a fresh (C, D) from $\mathcal{O}$. Obviously, this happens with a negligible probability $2 \cdot 2^{-n|\mathbb{F}|}$ only, so it has almost no impact on the efficiency of the protocol.

The reason for introducing Conditions 2 and 3 is to make the exposition simpler as it avoids dealing with the events that happen with negligible probability (cf. the caption of Figure 1). The reason for having Condition 1 is slightly more subtle and will be explained below.

The refreshing scheme is presented in Figure 1. The main idea behind this protocol is as follows (for this high-level overview ignore Step 4, as it anyway influences the execution only with negligible probability). Denote $\alpha := \langle A, B\rangle (= -\langle \tilde{A}, \tilde{B}\rangle)$. The Steps 2 and 3 are needed to refresh the share of P_{R}. This is done by generating, with the "help" of (A, B) (coming from $\mathcal{O}'$) a vector X such that

$$\langle L, X\rangle = \alpha. \tag{2}$$

Eq. (2) comes from simple linear algebra: $\langle L, X\rangle = \langle L, B \cdot M^T\rangle = \langle L \cdot M, B\rangle = \langle A, B\rangle = \alpha$. Then, vector X is added to the share of P_{R} by setting (in Step 3) $R' := R + X$. Hence we get $\langle L, R'\rangle = \langle L, R\rangle + \langle L, X\rangle = \langle L, R\rangle + \alpha$. Symmetrically, in Steps 5 and 6 the players refresh the share of P_{L}, by first generating Y such that $\langle Y, R\rangle = -\alpha$, and then setting $L' = L + Y$. By similar reasoning as before, we get $\langle L', R'\rangle = \langle L, R'\rangle - \alpha$, which, in turn is equal to $\langle L, R\rangle$. Hence, the refreshing is correct.

The security proof of this refreshing scheme appears in the full version of this paper. The key property that is used there is that X is generated "obliviously" from P_{L}, and Y is generated "obviously" from P_{R}. In other words: P_{L} gets no information on X except that $\langle L, X\rangle = -\langle Y, R\rangle$, and a symmetric fact holds for P_{R}. For more intuition behind this protocol the reader may consult [DF11] (Sect. 3), where a similar refreshing scheme is constructed. The main difference is that the protocol presented here refreshes the shares "completely", i.e. the new encoding (L', R') is completely independent from (L, R) (except that is encodes the same secret), while in [DF11] this was not the case. More precisely, in the refreshing of [DF11] $A, \tilde{A}, B$, and $\tilde{B}$ were such that $\langle A, B\rangle = \langle \tilde{A}, \tilde{B}\rangle = 0$, which implied that in particular $\langle L, R' - R\rangle$ and $\langle L' - L, R'\rangle$ were equal to 0 (and hence (L', R') was not independent from (L, R)). In our protocol it is not the case since $\langle A, B\rangle = \alpha$ and $\langle \tilde{A}, \tilde{B}\rangle = -\alpha$ (where α is random) and hence $\langle L, R' - R\rangle$ and $\langle L' - L, R\rangle$ are random. This "independence" of encodings after refreshing is a very useful property for showing security of composition of larger circuits.

Protocol $(L', R') \leftarrow \mathsf{Refresh}^n_{\mathbb{F}}((L, R))$**:**

Input (L, R)**:** $L \in (\mathbb{F} \setminus \{0\}) \times \mathbb{F}^{n-1}$ is given to P_{L} and $R \in \mathbb{F}^n$ is given to P_{R}.

1. Let $(A, \tilde{A}, B, \tilde{B}) \leftarrow \mathcal{O}'$ and give $(A, \tilde{A})$ to P_{L} and $(B, \tilde{B})$ to P_{R}.

Refreshing the share of P_{R}**:**

2. Player P_{L} generates a random non-singular matrix $M \in \mathbb{F}^{n\times n}$ such that $L \cdot M = A$ and sends it to P_{R}.
3. Player P_{R} sets $X := B \cdot M^T$ and $R' := R + X$.

Refreshing the share of P_{L}**:**

4. If $R' = (0, \dots, 0)$ then P_{R} sends a message $\mu =$ "`zero`" to P_{L}. Player P_{L} sets $L' \leftarrow (\mathbb{F} \setminus \{0\}) \times \mathbb{F}^{n-1}$. The players output (L', R') and finish this round of refreshing. Otherwise the player P_{R} sends a message $\mu =$ "`nonzero`" to P_{L} and they execute the following:
5. Player P_{R} generates a random non-singular matrix $\tilde{M} \in \mathbb{F}^{n\times n}$ such that $\tilde{M} \cdot R' = \tilde{B}$ and sends it to P_{L}.
6. Player P_{L} sets $Y := \tilde{A} \cdot \tilde{M}^T$ and $L' := L + Y$.
7. If $L'[1] = 0$ then restart the procedure of refreshing the share of P_{L}, i.e. go to Step 4.

Output: The players output (L', R').
Views: The view $\mathsf{view}_{\mathsf{L}}$ of player P_{L} is $(L, A, M, \tilde{A}, \tilde{M}, \mu)$ and the view $\mathsf{view}_{\mathsf{R}}$ of player P_{R} is $(R, B, M, \tilde{B}, \tilde{M}, \mu)$.

Fig. 1. Protocol $\mathsf{Refresh}^n_{\mathbb{F}}$. Oracle $\mathcal{O}'$ samples random vectors $(A, \tilde{A}, B, \tilde{B}) \in (\mathbb{F}^n)^4$ such that (1) $\langle A, B\rangle = -\langle \tilde{A}, \tilde{B}\rangle$ and (2) $A \neq (0)^n$, and (3) $\tilde{B} \neq (0)^n$. Note that the conditions (2) and (3) are needed as otherwise it might be impossible to find matrices M and $\tilde{M}$ in Steps 2 and 5, respectively. It is easy to see that $L[1]$ has a uniform distribution over $\mathbb{F}$, and hence restarting part of the protocol in Step 7 happens with probability $|F|^{-1}$. Therefore if $\mathbb{F}$ is large then this probability is negligible. In Sect. 6 we show how to change our protocol so that the probability of restarting is negligible even if $|\mathbb{F}|$ is small (e.g. constant).

4.2 Leakage-Resilient Computation of Generalized Multiplication

We now present a leakage-resilient protocol for computing a "generalized multiplication" function $f(S^0, S^1, c) = c - S^0 \cdot S^1$, where the values $S^0 \in \mathbb{F}$ and $S^1 \in \mathbb{F}$ are encoded by an LRS $\Phi^n_{\mathbb{F}} = (\mathsf{Encode}^n_{\mathbb{F}}, \mathsf{Decode}^n_{\mathbb{F}})$ (let (L^0, R^0) and

(L^1, R^1) be the respective encodings), and $c \in \mathbb{F}$ is a constant. The result $f(S^0, S^1, c)$ of the computation is encoded by (L'', R''). This construction has already been discussed informally in Section 3. The formal description appears in Figure 2. It uses the $\mathsf{Refresh}_{\mathbb{F}}^{n^2}$ protocol as a sub-routine, and hence also relies on the special free oracle $\mathcal{O}'$. It is easy to see that this protocol is correct. More formally, for any inputs $L^0, R^0, L^1, R^1 \in \mathbb{F}^n$ and $c \in \mathbb{F}$ we have that $\mathsf{Decode}_{\mathbb{F}}^n(L'', R'') = c - \mathsf{Decode}_{\mathbb{F}}^n(L^0, R^0) \cdot \mathsf{Decode}_{\mathbb{F}}^n(L^1, R^1)$, where $(L'', R'') \leftarrow \mathsf{Mult}_{\mathbb{F}}^n((L^0, R^0), (L^1, R^1), c)$. The security properties of this protocol are defined and proven in the full version of this paper, where we show that the multiplication protocol is reconstructible with low communication between the parties P_{L} and P_{R}.

Protocol $(L'', R'') \leftarrow \mathsf{Mult}_{\mathbb{F}}^n((L^0, R^0), (L^1, R^1), c)$**:**

Input (L, R)**:** $L^0, L^1 \in (\mathbb{F} \setminus \{0\}) \times \mathbb{F}^{n-1}$ are given to P_{L} and $R^0, R^1 \in \mathbb{F}^n$ are given to P_{R}. The field element $c \in \mathbb{F}$ is given to both players.

1. The players P_{L} and P_{R} run the $\mathsf{Refresh}_{\mathbb{F}}^{n^2}(L^0 \otimes L^1, R^0 \otimes R^1)$ protocol. Let L' and R' be their respective outputs, and let $\mathsf{view}'_{\mathsf{L}}$ and $\mathsf{view}'_{\mathsf{R}}$ be their respective views.
2. Player P_{L} sends $x := L'[1]$ and the last $n(n-1)$ bits of L' (i.e. the vector $L'[n+1, \ldots, n^2]$) to P_{R}. Player P_{R} computes $d := \langle L'[n+1, \ldots, n^2], R'[n+1, \ldots, n^2] \rangle$ and sets $R'' := -R'[1, \ldots, n] + (x^{-1}(c-d), 0, \ldots, 0)$.
3. Player P_{L} sets $L'' := L'[1, \ldots, n]$.

Output: The players output (L'', R'').
Views: The view $\mathsf{view}_{\mathsf{L}}$ of player P_{L} is $(L^0, L^1, L', L'', c, \mathsf{view}'_{\mathsf{L}})$ and the view $\mathsf{view}_{\mathsf{R}}$ of player P_{R} is $(R^0, R^1, R', R'', c, d, x, L'[n+1, \ldots, n^2], \mathsf{view}'_{\mathsf{R}})$.

Fig. 2. Protocol $\mathsf{Mult}_{\mathbb{F}}^n$. Note that computing x^{-1} is possible since in our LRS the first bit of L is never equal to 0. This is actually precisely the reason why this restriction was introduced.

5 The Compiler

5.1 Arithmetic Circuits

Before describing our general circuit compiler, we must define how to model arithmetic circuits over finite fields $\mathbb{F}$ as these are used to describe the original circuits. To keep the exposition simple, we consider circuits consisting only of 4 types of gates. The first two types are: the *public-input gates* that will be used by the user, or the adversary, to provide the input X to the circuit, and the *private-input gates* that will be used to provide the secret input *state* (e.g., the cryptographic key) to the scheme. The third type of a gate is the *multiplication gate* (a, b, c). This gate takes as input the values $A \in \mathbb{F}$ and $B \in \mathbb{F}$ of two other gates (indicated by a and b, resp.) and a constant $c \in \mathbb{F}$, and produces a result $c - AB$. Note that in particular the "negated and" function over bits can be expressed by such a gate, as $\overline{A \wedge B} = 1 - AB$, for $A, B \in \{0,1\}$. Finally, we also have the output gates. Each output gate takes as input a value from of a gate of a previous type and outputs it. Since it is well-known that a NAND gate is complete the above suffices to describe any functionality. Formally, a *circuit over a field* $\mathbb{F}$ is a sequence $\Gamma = (\gamma_1, \ldots, \gamma_t)$, where each γ_i is called a *gate.* The set of gates is divided into the following groups.

public-input gates: $\gamma_1, \ldots, \gamma_m$ — each such a gate is equal to a special symbol pub and takes the inputs provided by the user.

private-input gates: $\gamma_{m+1}, \ldots, \gamma_{m+k}$ — each such a gate is equal to a special symbol priv and represents the memory containing the secret state,

multiplication gates: $\gamma_{m+k+1}, \ldots, \gamma_{t-u}$ — each such a gate γ_i ($i \in [m+k+1, t-u]$) has a form (a, b, c), where $a, b \in \{1, \ldots, i-1\}$ and $c \in \mathbb{F}$. We say that the outputs of the gates γ_a and γ_b are *inputs for the gate* γ_i,

output gates: $\gamma_{t-u+1}, \ldots, \gamma_t$ — each such a gate γ_i is equal to some j, where $j \in \{1, \ldots, t-u\}$. We say that γ_j is an *input for the gate* γ_i.

For technical reasons, we also assume that the circuit's fan-out is at most 2, more precisely: each γ_i is an input for at most 2 other gates. This can be clearly done without loss of generality. The *computation* $\mathsf{Comp}(\Gamma, X, \mathit{state})$ of such a circuit on input $(X, \mathit{state}) = ((x^1, \ldots, x^m), (s^1, \ldots, s^k))$ is a sequence $(\xi^1, \ldots, \xi^t)$ of values on the outputs of circuit gates (one may think of this as the output wires of the gates), defined by the following procedure:

– For $i = 1$ to t do:
  1. if $\gamma_i = \mathsf{pub}$ ("public-input gate") then set $\xi^i := x^i$,
  2. if $\gamma_i = \mathsf{priv}$ ("private-input gate") then set $\xi^i := s^{i-m}$,
  3. if $\gamma_i = (a, b, c)$ ("multiplication gate") then set $\xi^i = c - \xi^a \xi^b$.
  4. if $\gamma_i = j$ ("output gate") then set $\xi^i = \xi^j$,

The *output of the computation* is equal to $(\xi^{t-u+1}, \ldots, \xi^t)$ and will be denoted by $\Gamma(X, \mathit{state})$.

5.2 Protocols Computing Circuits

Recall the definition of a protocol from Sect. 2.1. In this section we consider a special type of such protocols, that we call *LRS-protocols.* Each such a protocol Π_Φ is parameterized by an LRS $\Phi = (\mathsf{Encode} : \mathcal{M} \to \mathcal{L} \times \mathcal{R}, \mathsf{Decode} : \mathcal{L} \times \mathcal{R} \to \mathcal{M})$ (we will say that *Π works over Φ*). It consists of $2t$ parties $\mathcal{P} = \{P_\mathsf{L}^1, \ldots, P_\mathsf{L}^t, P_\mathsf{R}^1, \ldots, P_\mathsf{R}^t\}$. The parties are divided into following groups:

"public-input parties": $P_\mathsf{L}^1, \ldots, P_\mathsf{L}^m, P_\mathsf{R}^1, \ldots, P_\mathsf{R}^m$ — each P_L^i takes no input and each P_R^i takes as input $x^i \in \mathbb{F}$,

"private-input parties": $P_\mathsf{L}^{m+1}, \ldots, P_\mathsf{L}^{m+k}, P_\mathsf{R}^{m+1}, \ldots, P_\mathsf{R}^{m+k}$ — each P_L^i takes as input $L^i \in \mathcal{L}$, and each P_R^i takes as input $R^i \in \mathcal{R}$,

"multiplication parties": $P_\mathsf{L}^{m+k+1}, \ldots, P_\mathsf{L}^{t-u}, R^{m+k+1}, \ldots, P_\mathsf{R}^{t-u}$ — they have no inputs or outputs,

"output parties": $P_{\mathsf{L}}^{t-u+1}, \ldots, P_{\mathsf{L}}^{t}, R^{t-u+1}, \ldots, P_{\mathsf{R}}^{t}$ — each P_{R}^{i} produces an output $y^i \in \mathcal{M}$, and the P_{L}^{i}'s produce no output.

The LRS-protocols will be analyzed only under the assumption that for $i = k+1, \ldots, m$ we have that $(L^i, R^i) \leftarrow \mathsf{Encode}(z^i)$ for some x^i. More precisely for $X = (x^1, \ldots, x^m) \in \mathbb{F}^m$ and $state = (s^1, \ldots, s^k) \in \mathbb{F}^k$ consider the following experiment.

Experiment $\mathsf{ExpExec}(\Pi_{\Phi}, X, state)$**:**

1. For each $i = 1, \ldots, m$ give x^i to P_{R}^{i}.
2. For each $i = 1, \ldots, k$ sample $(L^{m+i}, R^{m+i}) \leftarrow \Phi(s^i)$. Give L^{m+i} to P_{L}^{m+i} and R^{m+i} to P_{R}^{m+i}.
3. Run the protocol Π_{Φ} with the inputs for the players as described in the previous steps.
4. For $i = 1, \ldots, t$ let $\mathsf{view}_{\mathsf{L}}^{i}$ be the view of P_{L}^{i}, and let $\mathsf{view}_{\mathsf{R}}^{i}$ be the view of P_{R}^{i} in the above execution.
   Denote $\mathsf{View}(\Pi_{\Phi}, (X, state)) := ((\mathsf{view}_{\mathsf{L}}^{1}, \mathsf{view}_{\mathsf{R}}^{1}), \ldots, (\mathsf{view}_{\mathsf{L}}^{t}, \mathsf{view}_{\mathsf{R}}^{t}))$.
5. Let $\mathsf{Exec}(\Pi_{\Phi}, (X, state))$ be the vector containing the outputs of the parties $P_{\mathsf{R}}^{t-u+1}, \ldots, P_{\mathsf{R}}^{t}$ in the above execution.

5.3 The Security Definition

We now present the main security definition of this paper. As mentioned in the introduction, in this definition we consider only the non-adaptive security. In Sect. 6 we show how this definition can be extended to adaptive settings. Let Γ be a circuit with m public-input gates, k private-input gates and u output gates. Let Π_{Φ} be an LRS-protocol with $2m$ public-input parties, $2k$ private-input parties and $2u$ output parties. We say that *the Π_{Φ} protocol (λ, ϵ)-securely* computes Γ if:

– *Π_{Φ} computes Γ* i.e.: for every $(X, state) \in \mathbb{F}^k \times \mathbb{F}^m$ we have that

$$\mathsf{Exec}(\Pi_{\Phi}, (X, state)) = \Gamma(X, state),$$

and

– for every λ-limited adversary $\mathcal{A}$ there exists a *simulator $\mathcal{S}$*, running in time polynomial in the running time of $\mathcal{A}$, that for every $(X, state) \in \mathbb{F}^k \times \mathbb{F}^m$, on input $(X, \Gamma(X, state))$ produces a variable $\mathcal{S}(X, \Gamma(X, state))$ such that

$$\Delta((\mathcal{S}(X, \Gamma(X, state))\ ;\ (\mathcal{A} \rightleftarrows \mathsf{View}(\Pi_{\Phi}, (X, \Gamma(X, state)))) \leq \epsilon. \qquad (3)$$

Note that *state* is not given directly to the simulator. The only variables that he gets are: the public input X and the output $Y = \Gamma(X, state)$. Therefore, intuitively, the only information that he gets about *state* comes from (X, Y).

5.4 The Construction

We are now ready to present our construction of the circuit compiler. Our compiler takes an arithmetic circuit Γ and a parameter $n \in \mathbb{N}$ and produces an LRS protocol $\Pi^{\Gamma}_{\Phi^n_{\mathbb{F}}}$ over $\Phi^n_{\mathbb{F}}$. To simplify the notation we will write Π^{Γ}_n instead of $\Pi^{\Gamma}_{\Phi^n_{\mathbb{F}}}$. The protocol Π^{Γ}_n is depicted on Fig. 3.

Protocol $(z^{t-u+1}, \ldots, z^t) \leftarrow (\Pi^{\Gamma}_n(x^1, \ldots, x^m, (L^1, R^1), \ldots, (L^k, R^k)))$**:**

Input $(x^1, \ldots, x^m, (L^1, R^1), \ldots, (L^k, R^k))$**:** Give each $x^i \in \mathbb{F}$ to P^i_{R}, each $L^i \in \mathbb{F}^n$ to P^{m+i}_{L} and each $R^i \in \mathbb{F}^n$ to P^{m+i}_{R}.

1. For $i = 1, \ldots, m$ player P^i_{R} computes $(L^i, R^i) \leftarrow \mathsf{Encode}^n_{\mathbb{F}}(x^i)$ and sends L^i to P^i_{L}. The view $\mathsf{view}^i_{\mathsf{L}}$ of P^i_{L} is L^i and the view $\mathsf{view}^i_{\mathsf{R}}$ of P^i_{R} is (L^i, R^i).
2. For $i = m+1, \ldots, m+k$ the view $\mathsf{view}^i_{\mathsf{L}}$ of P^i_{L} is L^i and the view $\mathsf{view}^i_{\mathsf{R}}$ of P^i_{R} is R^i.
3. For $i = m+k+1, \ldots, t-u$ let (a, b, c) be such that $\gamma_i = (a, b, c)$
   (a) Player P^a_{L} sends L^a to P^i_{L}.
   (b) Player P^a_{R} sends R^a to P^i_{R}.
   (c) Player P^b_{L} sends L^b to P^i_{L}.
   (d) Player P^b_{R} sends R^b to P^i_{R}.
   (e) Players P^i_{L} and P^i_{L} execute the $\mathsf{Mult}^n((L^a, R^a), (L^b, R^b), c)$ protocol. Let L^i and R^i be the respective outputs of the players at the end of this protocol, and let $\mathsf{view}^i_{\mathsf{L}}$ and $\mathsf{view}^i_{\mathsf{R}}$ be their respective views.
4. For $i = t-u+1, \ldots, t$ let j be such that $\gamma_i = j$.
   (a) Player P^j_{L} sends L^j to P^i_{L}.
   (b) Player P^j_{R} sends R^j to P^i_{R}.
   (c) The players P^j_{L} and P^i_{R} execute the $\mathsf{Refresh}^n(L^j, R^j)$ protocol. Let L^i and R^i be the respective outputs of the players at the end of this protocol, and let $\mathsf{view}^i_{\mathsf{L}}$ and $\mathsf{view}^i_{\mathsf{R}}$ be their respective views.
   (d) Player P^i_{L} sends L^i to P^i_{R}. Player P^i_{R} computes $z^j := \mathsf{Decode}^n_{\mathbb{F}}(L^i, R^i)$ and outputs it. The v^i of P^i_{L} is $\mathsf{view}^i_{\mathsf{L}}$ and the view $\mathsf{view}^i_{\mathsf{R}}$ of P^i_{R} is $(\mathsf{view}^i_{\mathsf{R}}, L^i)$.

Fig. 3. The Π^{Γ}_n protocol

We now have the following theorem. Its proof is based on the *hybrid argument* and appears in the full version of this paper.

Theorem 1. *Assume that for some n the LRS* $(\mathsf{Encode}^n_{\mathbb{F}}, \mathsf{Decode}^n_{\mathbb{F}})$ *is (λ, ϵ)-secure for some λ and ϵ. Then for any Γ the Π^{Γ}_n protocol $(\lambda/3 - \log_2 |\mathbb{F}|, t\epsilon)$-securely computes Γ.*

The following is an example of the application of Thm. 1 for a concrete LRS.

Corollary 2. *Suppose $|\mathbb{F}| = \Omega(n)$. Then for any Γ the Π^{Γ}_n protocol $(0.16 \cdot \log_2 |\mathbb{F}^n| - 1 - \log_2 |\mathbb{F}|, negl(n))$-securely computes Γ, for some negligible n.*

6 Extensions

The model in Sect. 5 was intentionally kept simple in order to make the proof as easy as possible, and to satisfy the page limit. In this section we present several generalizations and extensions of this model. The formal security definitions and proofs will be presented in the extended version of this paper.

Adaptive security. Most of the cryptographic security definitions assume that the adversary is *adaptive*, meaning that he can interact with the cryptographic device in rounds, and his queries in the ith round may depend on the answers that he got in rounds $1, \ldots, i-1$. Our model from Sect. 5 obviously does not cover this scenario. We now briefly argue how to extend the model and the protocol to cover also the adaptive security. In the adaptive model one assumes that the circuit Γ is initialized with some secret $state \in \mathbb{F}^k$ and it can be queried adaptively on several inputs $X^1, \ldots, X^\ell$ (where ℓ is the number of rounds). To each such a query the circuit responds with $Y^i := \Gamma(X^i, state)$. The input X^i is placed on the "private input gates" at the beginning of each round, and the output Y^i appears on the "output gates".

The protocol Π^Γ that "computes Γ" consists of $2t$ parties, whose role is exactly like in the protocol in Sect. 5. In particular: the "private input parties" are initialized with an encoding of *state*, the "public input parties" in the ith round take X^i as input, and the output Y^i is produced by the "output parties". After the end of each round the memory of all the parties (except the "private-input parties" that hold the encoding of *state*) gets erased. The adversary $\mathcal{A}$ can adaptively choose the X^i's and leak at most λ bits from each party in *each round* of the computation of Π^Γ on input X^i. The security definition assumes that for each round the simulator $\mathcal{S}$ gets a pairs $\{(X^i, Y^i)\}_{i=1}^\ell$ and his goal is to produce the output that is statistically close to the output of $\mathcal{A}$.

The implementation of Π^Γ is similar to the implementation of Π^Γ from Sect. 5. In particular, the protocols for the parties in a single round are the same as before. The only change is that, since *state* does not change between the rounds, the "private input parties" need to refresh the encodings that they hold. This can be done easily with the $\mathsf{Refresh}_{\mathbb{F}}^n$ protocol from Sect. 4.1: each pair $(P_{\mathsf{L}}^i, P_{\mathsf{R}}^i)$ of "private input parties" applies, at the end of each round, the refreshing protocol to their encoding $(L^{\cdot} R^i)$, setting $(L^i, R^i) := \mathsf{Refresh}_{\mathbb{F}}^n(L^i, R^i)$. The security proof goes along the same lines as the proof of Thm. 1. It will be provided in the extended version of this paper.

More general circuits. The circuits that we consider in Sect. 5 have a very restricted form in order to make the proof of Thm. 1 as simple as possible. We now argue how some of these restrictions can be avoided. First, observe that we can consider circuits with fan-out $q > 2$. The only price to pay is that the leakage bound in the statement of Thm. 1 changes from "$\lambda/3 - |\mathbb{F}|$" to "$\lambda/(q+1) - |\mathbb{F}|$". This is because now each (L^i, R^i) is given to at most $q+1$ parties (not just 3 parties as before).

For some applications it may also be useful to have a separate procedure for adding values in a leakage resilient way. First, observe that adding a publiclyknown constant c to an encoded secret can be done easily, as depicted on Fig. 4 (protocol $\mathsf{AddConst}_{\mathbb{F}}^{n}$). In fact, this protocol has already been described in Sect. 3 used (implicitly) in protocol $\mathsf{Mult}_{\mathbb{F}}^{n}$ (cf. Fig. 2, Step 2). The protocol computing the sum of two encoded secrets is presented on Fig. 4. Correctness of this protocols is a simple calculation. Because of the lack of space we the formal pro of their security properties is moved to the full version of this paper.

Protocol $(L', R') \leftarrow \mathsf{AddConst}_{\mathbb{F}}^{n}((L, R), c)$**:**

Input (L, R)**:** $L \in (\mathbb{F} \setminus \{0\}) \times \mathbb{F}^{n-1}$ is given to P_{L} and $c \in \mathbb{F}$ is given to both players.

1. Player P_{L} sends $x := L[1]$ to P_{R}.
2. Player P_{R} computes $\tilde{R} := R + (x^{-1} \cdot c, 0, \ldots, 0)$
3. The players execute the $\mathsf{Refresh}(L, \tilde{R})$ procedure. Let (L', R') be the result.

Output: The players output (L', R').

Protocol $(L', R') \leftarrow \mathsf{Add}_{\mathbb{F}}^{n}((L^0, R^0), (L^1, R^1))$**:**

Input (L, R)**:** $L^0, L^1 \in (\mathbb{F} \setminus \{0\}) \times \mathbb{F}^{n-1}$ are given to P_{L} and $R^0, R^1 \in \mathbb{F}^n$ are given to P_{R}.

1. Player P_{L} sets $A := L^0$ and $C := L^1 - L^0$.
2. Player P_{L} sets $B := R^0 + R^1$ and $D := R^1$.
   Note that $\langle A, B\rangle + \langle C, D\rangle = \langle L^0, R^0\rangle + \langle L^1, R^1\rangle$.
3. Refresh (C, D) by $(C', D') \leftarrow \mathsf{Refresh}_{\mathbb{F}}^{n}(C, D)$.
4. Compute $c := \mathsf{Decode}_{\mathbb{F}}^{n}(C', D')$.
   Note that this does not reveal any information about the inputs of the protocol, as (C', D') were "refreshed".
5. Set $(L', R') \leftarrow \mathsf{AddConst}_{\mathbb{F}}^{n}((A, B), c)$

Output: The players output (L', R').

Fig. 4. Protocols $\mathsf{AddConst}_{\mathbb{F}}^{n}$ and $\mathsf{Add}_{\mathbb{F}}^{n}$

Dealing with small fields. A natural field over which one could use our compiler is Z_2. The problem here is that we assumed that in our encoding we have $L[1] \neq 0$, and in the refreshing protocol, if this condition is not met, then part of the protocol is restarted (cf. Fig. 1). Of course if $\mathbb{F}$ is small then this restarting can happen with a high probability. To avoid this problem one could change the underlying encoding scheme and require that some prefix of L of length $a = \omega(\log_{|\mathbb{F}|}(n))$ (instead of just $L[1]$) is not equal to $(0)^a$. In this way the probability of restarting is at most $|\mathbb{F}|^{-a}$ and hence it is negligible in n. The other change that is also needed in this case is that in Step 2 of the $\mathsf{Mult}_{\mathbb{F}}^{n}$ protocol the player P_{L} needs to send $L[1, \ldots, a]$ (instead of $L[1]$) to P_{R}. The price to pay for it is that the "$- |\mathbb{F}|$" term in the leakage bound needs to be replaced by 2^a.

Smaller number of parties. Recall that the number of parties in the protocol Π^{Γ} corresponds to the number of independent memory parts in the real implementation of the scheme. In our model this number is linear ($2t$) in the number t of the gates of Γ. This can be reduced in the following way. First, observe that some parties can be "reused" if we look at the computation of Γ as a procedure that evaluates Γ gate-by-gate (cf. Sect. 5.1). More precisely: if a given gate γ^i is not used anymore as an input to other gates, then the memory of the party P^i that corresponds to γ^i can be erased and P^i can be "assigned" to some other gate. Hence, we can reduce the number of parties to $2t'$, where t' is the *width* of Γ. Here, by the "width" of a circuit we mean the minimal number of gates that needs to be kept in memory in order to compute Γ.

Observe also that we can actually decrease the number of memory parts even to two (call these parts: $\mathcal{L}$ and $\mathcal{R}$), by placing all P^i_{L}'s on $\mathcal{L}$, and all P^i_{R}'s on $\mathcal{R}$. This, however, comes at a price: the leakage bound of $\mathcal{L}$ and $\mathcal{R}$ still needs to be a constant fraction of $|n|$, and hence it is a $\frac{c}{t'} \cdot |\mathcal{L}|$ (where c is a constant and t' is the width of Γ), and the fraction $\frac{c}{t'}$ gets very small for large t'. Hence it is mostly of a theoretical interest.

Acknowledgments. The authors wish to thank Marcin Andrychowicz for pointing out some errors in an earlier version of this paper.

References

[DDV10] Davì, F., Dziembowski, S., Venturi, D.: Leakage-resilient storage. In: Garay, J.A., De Prisco, R. (eds.) SCN 2010. LNCS, vol. 6280, pp. 121–137. Springer, Heidelberg (2010)

[DF11] Dziembowski, S., Faust, S.: Leakage-Resilient Cryptography from the Inner-Product Extractor. In: Lee, D.H. (ed.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 702–721. Springer, Heidelberg (2011), `http://eprint.iacr.org/`

[DP08] Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: FOCS 2008: Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science. IEEE Computer Society, Washington, DC, USA (2008)

[FRR$^+$10] Faust, S., Rabin, T., Reyzin, L., Tromer, E., Vaikuntanathan, V.: Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 135–156. Springer, Heidelberg (2010)

[GKR08] Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: One-Time Programs. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 39–56. Springer, Heidelberg (2008)

[GR10] Goldwasser, S., Rothblum, G.N.: Securing Computation against Continuous Leakage. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 59–79. Springer, Heidelberg (2010)

[ISW03] Ishai, Y., Sahai, A., Wagner, D.: Private Circuits: Securing Hardware against Probing Attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Heidelberg (2003)

[JV10] Juma, A., Vahlis, Y.: Protecting Cryptographic Keys against Continual Leakage. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 41–58. Springer, Heidelberg (2010)

[KP10] Kiltz, E., Pietrzak, K.: Leakage Resilient ElGamal Encryption. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 595–612. Springer, Heidelberg (2010)

[MR04] Micali, S., Reyzin, L.: Physically Observable Cryptography (Extended Abstract). In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 278–296. Springer, Heidelberg (2004)

[Pie09] Pietrzak, K.: A Leakage-Resilient Mode of Operation. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 462–482. Springer, Heidelberg (2009)