Protecting Cryptographic Keys Against Continual Leakage

Ali Juma and Yevgeniy Vahlis*

Department of Computer Science, University of Toronto
{ajuma,evahlis}@cs.toronto.edu

Abstract. Side-channel attacks have often proven to have a devastating effect on the security of cryptographic schemes. In this paper, we address the problem of storing cryptographic keys and computing on them in a manner that preserves security even when the adversary is able to obtain information leakage during the computation on the key.
Using any fully homomorphic encryption with re-randomizable ciphertexts, we show how to encapsulate a key and repeatedly evaluate arbitrary functions on it so that no adversary can gain any useful information from a large class of side-channel attacks. We work in the model of Micali and Reyzin, assuming that only the active part of memory during computation leaks information. Our construction makes use of a single "leak-free" hardware token that samples from a distribution that does not depend on the protected key or the function that is evaluated on it. Our construction is the first general compiler to achieve resilience against polytime leakage functions without performing any leak-free computation on the protected key. Furthermore, the amount of computation our construction must perform does not grow with the amount of leakage the adversary is able to obtain; instead, it suffices to make a stronger assumption about the security of the fully homomorphic encryption.

1 Introduction

Leakage-resilient cryptographic constructions – constructions that remain secure even when internal state information leaks to the adversary – have received much recent interest. Traditionally, security models have treated such internal state information as perfectly hidden from the adversary. However, the development of various side-channel attacks has made it clear that this traditional view is inconsistent with physical reality. In a side-channel attack, an adversary obtains information about the internal state of a device by measuring such things as power consumption, computation time, and emitted radiation.

Cryptographic primitives with long term keys, such as encryption and signature schemes, are often targeted by such attacks. An adversary observing information leakage from computation on the key can potentially accumulate enough data over time to compromise the security of the scheme. Consequently, storing

* Supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

keys and computing on them in adversarial environments has been an important goal both in theory and practice. Indeed, many operating systems provide cryptographic facilities that allow programs to access keys only through designated functions, such as signing and encrypting. Smart cards provide a similar interface in hardware. In both cases, the goal is to limit any adversary to interacting with the scheme through designated channels. Nevertheless, information leakage through physical side-channels is often sufficient to overcome such barriers and break the scheme.

In this paper, we propose an approach for protecting cryptographic keys and computing on them repeatedly in a manner that preserves the secrecy of the key even when information about the state of the device continuously leaks to the adversary. Towards this goal, we define a new primitive called a *key proxy*, which encapsulates a key K and provides a structured way of evaluating arbitrary functions on K. This allows, for example, the conversion of any pseudorandom function, signature scheme, or public-key encryption scheme into a leakage-resilient variant of itself. Our construction withstands a bounded amount of leakage per invocation (where an invocation occurs each time a function is evaluated on K), but the total amount of leakage is unbounded. Previously, only stream ciphers, signature schemes, and identification scheme have been made resilient to an unbounded total amount of leakage.

For our construction, we make use of the recently achieved fully homomorphic encryption [12, 4], and an additional "leak-free" component. We describe two ways of instantiating this component, and in both cases the component samples from a globally fixed distribution that does not depend on K.

Leakage-resilient cryptography. The problem of executing code in an adversarial environment has always been on the minds of cryptographers. Still, most cryptographic schemes are designed assuming that the hardware on which they will be implemented is a black-box device, and information is accessible to the adversary only through external communication channels. Goldreich and Ostrovsky [13] consider the problem of protecting software from malicious users, and define the concept of an oblivious RAM – a CPU that is capable of evaluating encrypted programs using a constant amount of leak-free memory and an unbounded amount of memory that is fully visible to the adversary. The oblivious RAM is initialized with a secret key, which is used to decrypt encrypted instructions, execute them, and re-encrypt the output. The encrypted state of the program is stored in the clear. Oblivious RAMs provide the strong security guarantee that even if an adversary can keep track of the memory locations accessed by the computation, she is still unable to gain any additional information about the program over what would normally be revealed through black box access.

Since the work of Goldreich and Ostrovsky, the focus in leakage-resilient cryptography has been steadily shifting towards allowing the adversary ever-growing freedom in observing the *computation* of cryptographic primitives. Ishai, Sahai, and Wagner [17] introduce "private circuits" – a generic compiler that transforms any circuit into one that is resilient to probing attacks. In a probing attack, the

adversary selects a subset (of some fixed size) of the wires of the circuit and obtains the values of these wires. Goldwasser, Kalai, and Rothblum [15] define one-time programs – programs that come with small secure hardware tokens, and can be executed a bounded number of times without revealing anything but the output, even if the adversary observes the entire computation. The secure tokens are the hardware equivalent of oblivious transfer – each token stores two keys and reveals one of them upon request, while the second key is erased.

Micali and Reyzin [20] outline a framework for defining and analyzing cryptographic security against adversaries that perform side channel attacks. They introduce an axiom: only computation leaks information. That is, at any point during the execution of an algorithm, only the part of memory that is actively computed on may leak information. This allows for convenient modeling of leakage: an algorithm is described as a sequence of procedures and the set of variables that is accessed by the procedure. The adversary may then obtain leakage separately from the contents of each set of variables as they are accessed during the execution of the algorithm. The only-computation-leaks model (OCL) has since been used to obtain stream ciphers [9, 21] and signature schemes [10] that remain secure even if the adversary obtains leakage from the active state each time the primitive is used, and the total amount of leakage is unbounded. We refer to such leakage as "continuous leakage" for the rest of the paper.

Faust *et al* [11] propose an alternative restriction on side-channel adversaries: restricting the computational power of the leakage function but allowing leakage on the entire state. Faust *et al* describe a circuit transformation that immunizes any circuit against leakage functions that can be described as AC^0 circuits[1]. The transformed circuit can leak information from the entire set of wires at each invocation, and makes use of a polynomial number of leak-free components that generate samples from a fixed distribution that does not depend on the computation of the circuit. We make use of a similar leak-free component, although the distribution generated by our component is significantly more complex than the one in [11] due to the fact that we must defend against leakage functions that are not restricted to circuits of small depth.

Very recently, specific leakage-resilient cryptographic primitives have been constructed under even more general continuous leakage models. Dodis, Haralambiev, Lopez-Alt, and Wichs [7] have constructed several primitives, including signature schemes and authenticated key agreement protocols, that remain secure even if the entire state (and not just the active part) leaks information continuously. The public key of the scheme remains fixed throughout the lifetime of the system. Brakerski, Kalai, Katz, and Vaikuntanathan [3] construct a public-key encryption scheme that allows continuous leakage on the entire state, and does not require a leak-free key update procedure. Brakerski *et al* also construct signature schemes and identity based encryption under slightly different leakage models. As in our work, both above works provide protection against leakage that can be described by arbitrary polynomial-time computable functions with sufficiently short output.

[1] AC^0 circuits have constant depth and unbounded fan-in.

In addition to the recent work on cryptographic constructions that are resilient to continuous leakage, there has been significant progress [1, 2, 22, 19] on obtaining resilience to "memory attacks" – side channel attacks where the adversary obtains a bounded amount of information about the memory contents of the device throughout its lifetime. Perhaps due to the bounded nature of this type of leakage, constructions secure against memory attacks tend to be quite efficient and do not require the algorithm to maintain a state.

Concurrent work of Goldwasser and Rothblum. In a concurrent paper [16], Goldwasser and Rothblum construct a general compiler that achieves resilience to polynomial-time leakage. Their construction relies on a linear number of leak-free components, while ours relies on a single component. On the other hand, they rely on the standard Decisional Diffie Hellman assumption, whereas we rely on fully homomorphic encryption.

On testable leak-free components. When constructing leakage-resilient cryptographic primitives, one has to take care in the nature and amount of components that are assumed not to leak any information. It is preferable, but may not always be possible, to avoid such components altogether. For example, one can protect any functionality against leakage given an arbitrary number of leak-free gates that can decrypt a ciphertext, perform a logical operation on the plaintext, and re-encrypt the result. Such a component can be used to evaluate the circuit F on K gate by gate, keeping all intermediate values encrypted, and thereby rendering leakage useless. However, building such leak-free components may be as difficult as constructing a leak-free computer and forgetting all about side-channels. Consequently, the focus of research in this area has always been to reduce the power and amount of computation that is assumed to be a-priori insulated from side-channel attacks.

Our construction uses a leak-free component that produces random encryptions of some fixed message (in our case – $\bar{0}$) under a given public key in the fully homomorphic encryption scheme. More specifically, the leak-free component we use is a randomized component that, given *pub*, produces two random encryptions of $\bar{0}$. Consequently, the computation performed by this component does not depend on any user or adversarially supplied inputs, and in particular does not depend on the key K or the function F that is evaluated on K. We call such a component *testable* because it can be accurately simulated in a controlled environment – all one has to do is feed the component random bits and randomly generated public keys and observe its behavior. More generally, we say that a component is testable if its inputs come from a globally fixed distribution that is independent from other inputs to the system.

We propose testability as a rule of thumb for secure hardware components in leakage resilient cryptography. All hardware components leak at least *some* information such as timing (every computation takes time) and power consumption. Therefore, the best we can hope for is that the information leaked by the components that we assume to be leak-free is useless to the adversary. Testability gives us the ability to observe the leakage from the secure component – as it

will happen during actual usage – and estimate whether the component is safe to use. We note that the components used by [11] and [16] are testable.

In contrast to previous general compilers that achieve leakage resilience, we use only one leak-free component, regardless of the size of the circuit that is evaluated on K, or the amount of information leakage per invocation. Thus, our construction does not require the number of leak-free components to grow with the amount of leakage.

Our contributions. We study the problem of computing on a cryptographic key in an environment that leaks information each time a computation is performed. We show that in the OCL model with a single leak-free randomized token, a cryptographic key can be protected in a manner that allows repeated computation on it while making sure that the adversary gains no information from side-channel information leakage.

More precisely, we propose a tool which we call a *key proxy* – a stateful cryptographic primitive that is initialized once with a key K, and then given any circuit F computes $F(K)$. Any leakage obtained by an adversary from the computation of the key proxy can be computed given just F and $F(K)$. Using any *fully homomorphic encryption* (FHE) we construct a key proxy with the following properties:

Resilience to adaptive polynomial-time leakage. During each invocation of the key proxy, we allow the adversary to adaptively select leakage functions that are modeled as arbitrary circuits with a sufficiently short output. The exact amount of round leakage that our construction can withstand depends on the level of security of the underlying FHE. Assuming the most basic security for the FHE (i.e. against polynomial-time adversaries) permits security against $O(\log n)$ bits of leakage each time a function is evaluated on K. More generally, given a $2^{l(n)}$-secure FHE, our construction can withstand roughly $l(n)$ bits of leakage per invocation.

Independent complexity. The starting point of leakage-resilient cryptography is that *computation leaks information*. It does not require a large leap of faith to suspect that *more* computation leaks more information. In fact, to the best of our knowledge, this is indeed the case for many side-channel attacks in practice. The amount of computation performed by our key proxy construction does not depend on the amount of leakage that the adversary obtains per invocation. Instead, to get resilience to larger amounts of leakage, a stronger assumption about the security of the underlying fully homomorphic encryption is used. This allows us to avoid a circular dependency where, in order to obtain resilience to larger amounts of leakage one must build a more complex device, which in turn leaks more information.

One-time programs with efficient refresh. The one-time programs of [15] can be implemented without leak-free one-time memory tokens by storing the contents of the tokens in memory, and then accessing only the needed values during computation. The one-time programs can then be refreshed occasionally in a secure environment to allow continuous use. Currently, the refresh procedure performs as much computation as the evaluation of the program that it pro-

tects. If one is willing to trade resilience against complete exposure of the active memory (achieved by [15]) for resilience against length-bounded leakage then by pre-computing the outputs of the leak-free tokens in our construction and storing them in memory, we obtain one-time programs with an update procedure of fixed complexity that does not depend on the protected program.

Our approach. The underlying building block for our construction is fully homomorphic encryption. An FHE is a public-key encryption scheme that allows computation on encrypted data. That is, given a ciphertext with corresponding plaintext M, the public key, and a circuit F, there is an efficient algorithm that computes an encryption of $F(M)$.

For our construction, we partition the state of the key proxy into two parts, A and B (or, equivalently, two devices). Given a key K, the key proxy is initialized as follows. An FHE key pair (pri, pub) is generated and is stored in memory A. Then, a random encryption C of K under pub is computed and is stored in memory B. To evaluate a function F (described as a circuit) on K, the following actions are performed. First, a new pair of keys (pri', pub') is generated and stored in memory A, and an encryption $C_{pri} = \mathsf{Enc}_{pub'}(pri)$ of the old private key is written to a public channel. Then, computing on memory B and the public channel, the following two ciphertexts are generated homomorphically from C and C_{pri}: an encryption C_{res} of $F(K)$ and a fresh encryption C_{key} of K. Note that both C_{res} and C_{key} are encryptions under the new public key pub'. The ciphertext C_{res} is then sent back to memory A where it is decrypted, and $F(K)$ is returned as the output of the program. This basic approach is described in Figure 1.

It is clear that without leakage, the above construction is secure. Of course, the main difficulty is showing that leakage does not provide the adversary with any useful information. Below we provide an informal description of two main technical issues that arise.

Leakage on private keys and ciphertexts. It is easy to see that without refreshing the encryption C of K, a leakage adversary will eventually learn all of K by gradually leaking all of C and pri and then simply decrypting. Therefore, it is clear that an update procedure is necessary. The algorithm described in Figure 1 performs such an update: After each invocation, memory A contains a freshly generated private key and memory B contains an encryption of K under the corresponding public key. However, we cannot directly claim that this refreshing procedure provides the necessary level of security. The main difficulty stems from the fact that the adversary obtains leakage on the private key in memory A both before and after she obtains leakage on the encryption C of K under the corresponding public key. In particular, if the adversary could obtain the entire ciphertext C, she would be able to hardcode it into the second leakage function that is applied to the private key. The leakage function would then decrypt C and leak bits of information about K.

This requires us to make use of the fact that the adversary obtains only a bounded amount of leakage on the ciphertext C, and never sees it completely.

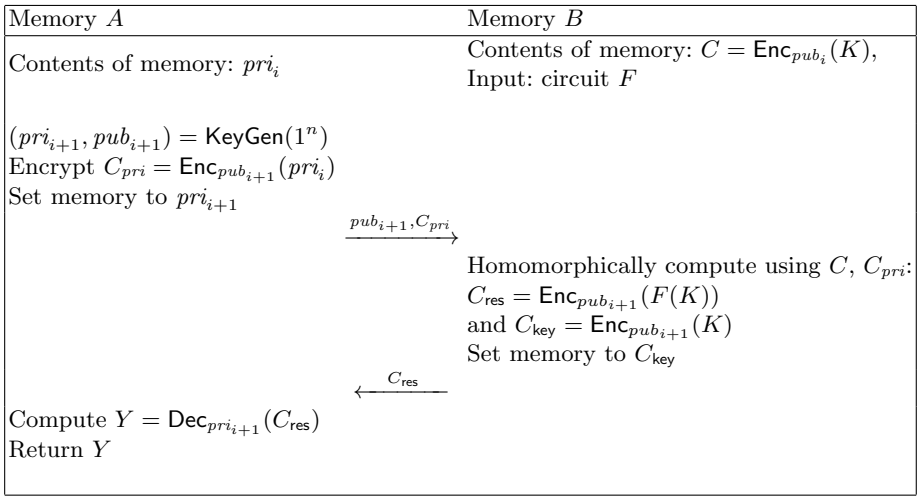

Fig. 1. Informal description of the construction

We argue that any leakage function that provides enough information about the ciphertext in order to later learn something about the plaintext given the private key, essentially acts as a distinguisher and can be used to break the semantic security of the FHE.

Randomizable FHE. Ciphertexts produced by fully homomorphic encryption schemes may carry information about the homomorphic computation that was performed to obtain them. For instance, it is possible that the ciphertext C_{res} is actually first decrypted to a string of the form $(F(K), K)$ and then the decryption algorithm ignores the second element in the pair. In this case, the adversarial leakage function is clearly not forced to follow the honest decryption algorithm and can make use of the intermediate values of the decryption process to leak information about K. Similarly, the ciphertext C_{key} may contain information about the function F that was evaluated on K. For some applications, such as encryption where F encodes in plain text the message to be encrypted, this is undesirable since the adversary may use future leakage functions to gain information about the message.

Fortunately, the homomorphic encryption schemes of Gentry [12] and of van Dijk *et al* [4] have the following additional property: given any encryption C of a message M and a random encryption C' of M', the ciphertext $C + C'$, where the addition is performed over the appropriate group of ciphertexts, is a random encryption of $M + M'$. Consequently, to address the issue described above, we randomize both C_{res} and C_{key} by adding random encryptions of zero to both ciphertexts. In order to make use of the property described above, the encryptions of zero need to be generated without leakage; otherwise, the leaked

information maintains a correlation between the randomized ciphertext and the history of the computation that was used to produce the original ciphertext.

We note that in the FHE schemes of [12] and [4], C' has to be generated in a special way in order to have enough noise to annihilate any dependence between $C + C'$ and the computation history of C. For simplicity of exposition we ignore this distinction, and instead remark that the randomization procedures of both FHE schemes satisfy the properties needed for our construction.

Function privacy in key proxies. In the above description of key proxies, we require that the leakage obtained by the adversary can be simulated given just F and $F(K)$. However, in some applications, such as private-key encryption, the function F itself also needs to be hidden. In the case of encryption, F contains the message M, so an adversary can break semantic security simply by leaking information about F, ignoring K completely. This raises a subtle modeling issue: the message M must exist somewhere as plaintext, and if the adversary obtains leakage on that computation, she will trivially break semantic security. Therefore, irrespective of the definition of leakage-resilient key proxies, semantic security cannot be achieved when every invocation of every algorithm leaks information.

There are several ways in which this issue can be addressed. One solution is to weaken the definition of semantic security by requiring that the plaintexts have high pseudo-entropy[2] given the leakage obtained by the adversary. We avoid this approach both because it leads to complex definitions, and because it does not seem to have a clear advantage over the following much cleaner solution. Instead, we allow the adversary to obtain leakage both before and after the challenge ciphertext is generated, but not on the computation of the challenge ciphertext itself. This essentially means that while leakage can compromise individual encryptions, the long-term key remains safe. Under this restriction, our definition of key proxies provides the needed level of security. This approach is consistent with previous definitions of leakage-resilient semantic security (see e.g. [9, 22, 8, 6]), and allows us to avoid additional complexity in our definition. This is desirable especially given the fact that for some applications of key proxies, such as signature schemes, function privacy is not necessary.

We mention briefly that another option is to define a leakage model for private-key encryption which allows the encryption algorithm to perform some leak-free pre-processing that is independent of the key. Then, the encryptor can generate an encrypted version of the circuit F, which can be safely given to the adversary without compromising security.

Organization. In Section 3, we describe the computational and leakage models that we use, and define a leakage-resilient key proxy. In Section 4, we provide our main construction, and analyze its security. In Section 5, we describe several variants of our model and construction, and provide some applications of leakage-resilient key proxies.

[2] A distribution has pseudo-entropy $\geq k$ if it is computationally indistinguishable from some distribution with min-entropy $\geq k$.

2 Preliminaries

Notation. We write PPT to denote Probabilistic Polynomial Time. When we wish to fix the random bits of a PPT algorithm M to a particular value, we write $M(x;r)$ to denote running M on input x and randomness r. We write $time_n(M)$ to denote the running time of algorithm M on security parameter n. We use $x \in_R S$ to denote the fact that x is sampled according to a distribution S. Similarly, when describing an algorithm we may write $x \leftarrow_R S$ to denote the action of sampling an element from S and storing it in a variable x.

It is common in cryptography to describe probabilistic experiments that test the ability of an adversary to break a primitive. Given such an experiment Exp, and an adversary A, we write $A \leftrightarrows \mathsf{Exp}$ to denote the random variable representing outcome of Exp when run with the adversary A.

2.1 Fully Homomorphic Encryption

The main tool in our construction is a fully homomorphic public-key encryption (FHE) system. Intuitively, such a system has the usual semantic security properties of a public-key encryption (PKE) scheme, but in addition, can perform arbitrary computation on encrypted data. The outcome of this computation is of course also encrypted. The first construction of FHE was given by Gentry in [12], and is based on ideal lattices. Recently another construction was proposed by van Dijk *et al* [4].

We do not go into the details of the FHE constructions, but rather present the result with respect to an arbitrary FHE with an additional randomization property, which is satisfied by both constructions.

Definition 1. *Let* $\mathsf{FHE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{EncEval}, \mathsf{Add}, \mathsf{Subtract})$ *be a tuple of PPT algorithms, and let* $l : \mathbb{N} \to \mathbb{N}$. *We say that* HPKE *is an* $l(n)$-secure fully homomorphic public key encryption scheme *if the following conditions hold:*

1. *The triple* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is a public-key encryption scheme. We assume without loss of generality that the private key is always the random bits of* KeyGen.
2. *The algorithm* $\mathsf{EncEval}(pub, \boldsymbol{C}, F)$, *where pub is a public key,* $\boldsymbol{C} = (C_1, \ldots, C_n)$ *is a vector of ciphertexts with plaintexts* $(m_1, \ldots, m_n)$, *and* F *is a circuit on* n *inputs, outputs a string* C' *which is a valid encryption of* $F(m_1, \ldots, m_n)$.
3. *The algorithms* Add *and* $\mathsf{Subtract}$ *have the following properties:*
   (a) *For all pri, for* $pub = \mathsf{KeyGen}(pri)$, *for all messages* M_1 *and* M_2, *for a random encryption* C_1 *of* M_1 *under pub and for every encryption* C_2 *of* M_2 *under pub,* $\mathsf{Add}(pub, C_1, C_2)$ *is distributed identically to* $\mathsf{Enc}_{pub}(M_1 + M_2)$, *and* $\mathsf{Subtract}(pub, C_1, C_2)$ *is distributed identically to* $\mathsf{Enc}_{pub}(M_1 - M_2)$.
   (b) *For all ciphertexts* C_1 *and* C_2, $\mathsf{Add}(pub, \mathsf{Subtract}(pub, C_2, C_1), C_1) = C_2$. *That is, subtracting a ciphertext is the inverse of adding it.*
4. *For every probabilistic adversary* A *running in time at most* $l(n)$, *the advantage of* A *in breaking the semantic security of* FHE *is at most* $1/l(n)$.

Remark 1. The algorithms Add and Subtract may be implemented as addition and subtraction over the space of ciphertexts, though we do not require this. In some fully homomorphic encryption schemes, Add and Subtract may not achieve the exact requirement of step 3 above. Specifically, Add and Subtract may produce an encryption that cannot be computed on homomorphically using EncEval. We note that this is not a problem for our construction since we only use EncEval on encryptions of *pri*, which are ephemeral and never the output of Add or Subtract. We avoid formalizing this issue to improve exposition.

3 Models and Definitions

In this section, we present the definition of a leakage-resilient key proxy (LRKP). We start with a syntactic description of the primitive, and then describe the security experiment and the leakage model.

Stateful Algorithms. Due to the continuous nature of side-channel attacks, it is necessary for an LRKP to maintain a state in order to achieve security. We model stateful algorithms by considering algorithms with a special input and output structure. A stateful randomized algorithm takes as input a triple $(x; R, S)$ where x is the query to the algorithm, R is a random string, and S is a state (when R is clear from context we omit it, and denote the input by $(x; S)$). It then outputs (y, S_{new}) where y is the reply to the query, and S_{new} is the new state.

Definition 2. *A* key proxy *is a pair* $KP = (\mathsf{KPInit}, \mathsf{KPEval})$*, where* KPInit *is an algorithm, and* KPEval *is a stateful algorithm. For fixed* $c \in \mathbb{N}$ *and for all* $n \in \mathbb{N}$*,* $K \in \{0,1\}^{n^c}$*,* $\mathsf{KPInit}(1^n, K)$ *outputs an initial state* S*. For every circuit* $F : \{0,1\}^{|K|} \to \{0,1\}^n$*, and random coins* R*, the stateful algorithm* $\mathsf{KPEval}(1^n, F; R, S)$ *outputs* $F(K)$*.*

We now describe the security experiment of LRKPs. This experiment is parameterized by the leakage structure on a single invocation of the KPEval algorithm. However, for clarity we start with the description of the general experiment, and then provide details on the leakage that occurs at each invocation. We model the the leakage resilience of a key proxy by requiring the leaked information to be simulatable. That is, we require the existence of a simulator Sim that, given F and $F(K)$, can simulate the leakage and messages obtained by the adversary during the computation of $\mathsf{KPEval}(1^n, F; R, S)$. No efficient adversary should be able to tell whether she is getting actual leakage and messages, or interacting with a simulator. We now describe the real and ideal security experiments:

Let $KP = (\mathsf{KPInit}, \mathsf{KPEval})$ be a key proxy. Let A and Sim be PPT algorithms, $n \in \mathbb{N}$, and consider the following two experiments:

ExpReal **(Real Interaction).** The interaction of the adversary with the key proxy proceeds as follows:

1. A key K is chosen by the adversary, and $\mathsf{KPInit}(1^n, K)$ is used to generate an initial state S.

2. The adversary repeats the following steps an arbitrary number of times:
   (a) The adversary submits a circuit F, which is evaluated on K by KPEval. During the computation, the adversary acts as a single invocation leakage adversary (described below in Definition 5) for KPEval.
   (b) At the end of the computation of KPEval, the adversary is given $F(K)$.
3. After the adversary is done making queries, it outputs a bit b.

$\mathsf{ExpIdeal}$ **(Ideal Interaction).** The interaction of the adversary with simulated leakage proceeds as follows:

1. The adversary submits a key K, which is not revealed to the simulator.
2. The adversary then repeats the following steps an arbitrary number of times:
   (a) The adversary submits a circuit F, and Sim is given F and $F(K)$. The adversary then acts as a single invocation leakage adversary according to Definition 5, except that the leakage functions are submitted to the simulator, which returns simulated leakage values and messages.
   (b) Eventually the adversary stops submitting leakage functions, and is given $F(K)$.
3. After the adversary is done making queries, it outputs a bit b.

Definition 3. *We say that* KP *is a* Leakage-Resilient Key Proxy *if for every PPT A there exists a PPT S and a negligible function $neg(\cdot)$ such that*

$$|\Pr[(A \leftrightarrows \mathsf{ExpReal}) = 1] - \Pr[(A \leftrightarrows \mathsf{ExpIdeal}) = 1]| \leq neg(n)$$

The above definition describes the security of an LRKP relative to some unspecified procedure which allows the adversary to obtain leakage during each invocation of KPEval. The exact procedure for a single-invocation leakage depends on the leakage model and on the structure of the implementation of KPEval. Below we formalize the structure of our solution, and describe the leakage obtained by the adversary during a single invocation of KPEval.

Our construction of KPEval is described as a protocol between two parties EvalA and EvalB that leak information separately, and where the messages between EvalA and EvalB are public. In this format, our construction requires two flows between the parties: one from EvalA to EvalB and one from EvalB to EvalA. The following definition formalizes this structure.

Definition 4. *A 2-round* split state key proxy $KP = (\mathsf{KPInit}, \mathsf{KPEval})$ *is a key proxy such that the state S is represented as a pair $S = (\mathsf{MemA}, \mathsf{MemB}) \in (\{0,1\}^{n^d})^2$ for some fixed $d \in \mathbb{N}$, and the algorithm* KPEval *is described as four algorithms* $(\mathsf{LeakFree}, \mathsf{EvalA}_1, \mathsf{EvalB}, \mathsf{EvalA}_2)$*, each running in time polynomial in n, where*

1. EvalA_1 *takes as input* MemA*,* OutLF_A*, and randomness* RandA*, and outputs an updated state* $\mathsf{MemA}' \in \{0,1\}^{n^d}$ *and a message M_{AB} to* EvalB*.*

2. $\mathsf{LeakFree}$ *takes as input message* M_{AB} *and randomness* RandLF*, and outputs string* OutLF*.*
3. EvalB *takes as input* MemB*,* RandB*,* OutLF*, the message* M_{AB}*, and a circuit* $F : \{0,1\}^{|K|} \to \{0,1\}^n$ *of arbitrary size. It then outputs an updated state* $\mathsf{MemB}' \in \{0,1\}^{n^d}$ *and a message* M_{BA} *to* EvalA*.*
4. EvalA_2 *takes as input* MemA'*, the message* M_{BA} *and outputs an updated state* MemA'' *and the result* $F(K)$*.*

The output of KPEval *is* $F(K)$*, and the updated state is* $(\mathsf{MemA}'', \mathsf{MemB}')$*.*

Recall that our construction requires a leak-free component. This leak-free component is modeled by algorithm $\mathsf{LeakFree}$ above. A crucial point here is that $\mathsf{LeakFree}$ receives only randomness and a public message as input, and, in particular, receives neither F nor the saved state $(\mathsf{MemA}, \mathsf{MemB})$ as inputs; therefore, regardless of the actual construction, the above definition prevents $\mathsf{LeakFree}$ from carrying out the evaluation of F on K, which would make the construction trivial.

We are now ready to describe the leakage structure on a single invocation of a 2-round split state key proxy. The leakage model we use, commonly known as "only computation leaks information" (OCL), lets the adversary obtain leakage only on the active part of memory during each computation.

Definition 5. *Let* $l : \mathbb{N} \to \mathbb{N}$ *and let KP be a 2-round split state key proxy. A single invocation leakage adversary in the only-computation-leaks model chooses a circuit* f_1*, then sees* $f_1(\mathsf{MemA}, \mathsf{RandA})$ *and* M_{AB}*, chooses circuit* f_2*, then sees* $f_2(\mathsf{MemB}, \mathsf{OutLF}, \mathsf{RandB})$ *and* M_{BA}*, chooses a circuit* f_3*, and finally sees* $f_3(\mathsf{MemA}')$*. The adversary is* l*-bounded if for all* n *the range of* f_1, f_2, f_3 *is* $\{0,1\}^{l(n)}$*.*

Note that in the above definition, the leakage functions can compute any internal values that appear during the computations of EvalA_1, EvalB, and EvalA_2. This means, for example, that it is unnecessary to explicitly provide M_{AB} to f_1 or M_{BA} to f_2.

History freeness. In Definition 3 we allow information about the functions F_i that are evaluated on K to leak to the adversary. In particular, it is possible that during some invocation j the adversary can obtain, through leakage, information about some previously queried function F_i. In the introduction we mentioned that leakage-resilient variants of some applications, such as private-key encryption, are defined to allow leakage both before and after the generation of the challenge ciphertext, but not on the challenge itself. However, if the state of LRKP keeps a history of some of the functions that were applied to K, then by leaking on it after the challenge was computed, the adversary may be able to break the semantic security of the encryption. We note that the above definition is sufficient to obtain security in the presence of what we call "lunch-time leakage" attacks – where the adversary obtains leakage only before the challenge ciphertext is generated, but not after.

To address the above issue, and allow full leakage in applications such as encryption, we introduce an additional information-theoretic property that requires that the state of the LRKP is distributed identically after all sequences of functions that are evaluated on K. This property is satisfied by our construction, and prevents the above mentioned "history attack".

Definition 6. *An LRKP* $(\mathsf{KPInit}, \mathsf{KPEval})$ *is called* history free *if for all* $n \in \mathbb{N}$ *and all* $K \in \{0,1\}^{poly(n)}$*, there exists a distribution* D *over the states of the LRKP such that for all* $j \in \mathbb{N}$*, all sequences of functions* $F_1, \ldots, F_j : \{0,1\}^{|K|} \to \{0,1\}^n$*, and all sequences of random tapes* $R_0, \ldots, R_{j-1}$*, the random variable* $\{S_{j+1} | S_1, \ldots, S_j\}$ *over* R_j *is distributed according to* D*, where* $S_1 = \mathsf{KPInit}(1^n, K; R_0)$ *and* S_i *is the updated state after* $\mathsf{KPEval}(1^n, F_{i-1}; R_i, S_{i-1})$*.*

4 Leakage-Resilient Key Proxies From Homomorphic Encryption

Given a fully homomorphic public-key encryption scheme $\mathsf{FHE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{EncEval}, \mathsf{Add}, \mathsf{Subtract})$ we construct a leakage-resilient 2-round split state key proxy $\mathsf{LRKP} = (\mathsf{KPInit}, \mathsf{KPEval})$.

$\mathsf{KPInit}(1^n, K)$: The algorithm $\mathsf{KPInit}(1^n, K)$ first runs $\mathsf{KeyGen}(1^n)$ to obtain a public-private key pair (pub_1, pri_1) for the FHE. It then generates a ciphertext $C_{\mathsf{key}} = \mathsf{Enc}_{pub_1}(K)$ and assigns $\mathsf{MemA} \leftarrow pri_1$ and $\mathsf{MemB} \leftarrow C_{\mathsf{key}}$. The output is an initial state that consists of two parts $(\mathsf{MemA}, \mathsf{MemB})$.

$\mathsf{KPEval}(1^n, F; (\mathsf{MemA}, \mathsf{MemB}))$: The algorithm KPEval consists of four subroutines: $\langle \mathsf{LeakFree}, \mathsf{EvalA}_1, \mathsf{EvalB}, \mathsf{EvalA}_2 \rangle$ that are used as follows: on input circuit F first generate $(\mathsf{OutLF}_A, \mathsf{OutLF}_B) \leftarrow_{\mathrm{R}} \mathsf{LeakFree}(1^n)$. Then, follow the protocol described in Figure 2 by computing

$$
\begin{aligned}
&(M_{AB}, \mathsf{MemA}') \leftarrow_{\mathrm{R}} \mathsf{EvalA}_1(\mathsf{MemA}, \mathsf{OutLF}_A);\\
&(M_{BA}, \mathsf{MemB}') \leftarrow_{\mathrm{R}} \mathsf{EvalB}(\mathsf{MemB}, \mathsf{OutLF}_B, M_{AB});\\
&Y \leftarrow \mathsf{EvalA}_2(\mathsf{MemA}', M_{BA})
\end{aligned}
$$

The final state after one evaluation of KPEval is $(\mathsf{MemA}', \mathsf{MemB}')$, and the output is Y.

We now describe the subroutines $\langle \mathsf{LeakFree}, \mathsf{EvalA}_1, \mathsf{EvalB}, \mathsf{EvalA}_2 \rangle$ of KPEval:

$\mathsf{LeakFree}(pub)$: Parse randomness as (r_{LF1}, r_{LF2}), and compute

$$
\begin{aligned}
C_{\mathsf{R0}} &= \mathsf{Enc}_{pub}(\bar{0}; r_{LF1})\\
C_{\mathsf{R1}} &= \mathsf{Enc}_{pub}(\bar{0}; r_{LF2})\\
\mathsf{OutLF} &= (C_{\mathsf{R0}}, C_{\mathsf{R1}})
\end{aligned}
$$

and output OutLF.

The subroutines EvalA_1, EvalB, and EvalA_2 are described in Figure 2 as a two round two party protocol where EvalA_1 and EvalA_2 specify the actions of party A and EvalB specifies the actions of party B. In the definition of EvalB we use subroutines $\mathsf{Evaluate}$ and $\mathsf{Refresh}$ that are defined as follows:

$\mathsf{Evaluate}(F, C, pri)$: Compute and output $F(\mathsf{Dec}_{pri}(C))$
$\mathsf{Refresh}(C, pri)$: Compute and output $\mathsf{Dec}_{pri}(C)$

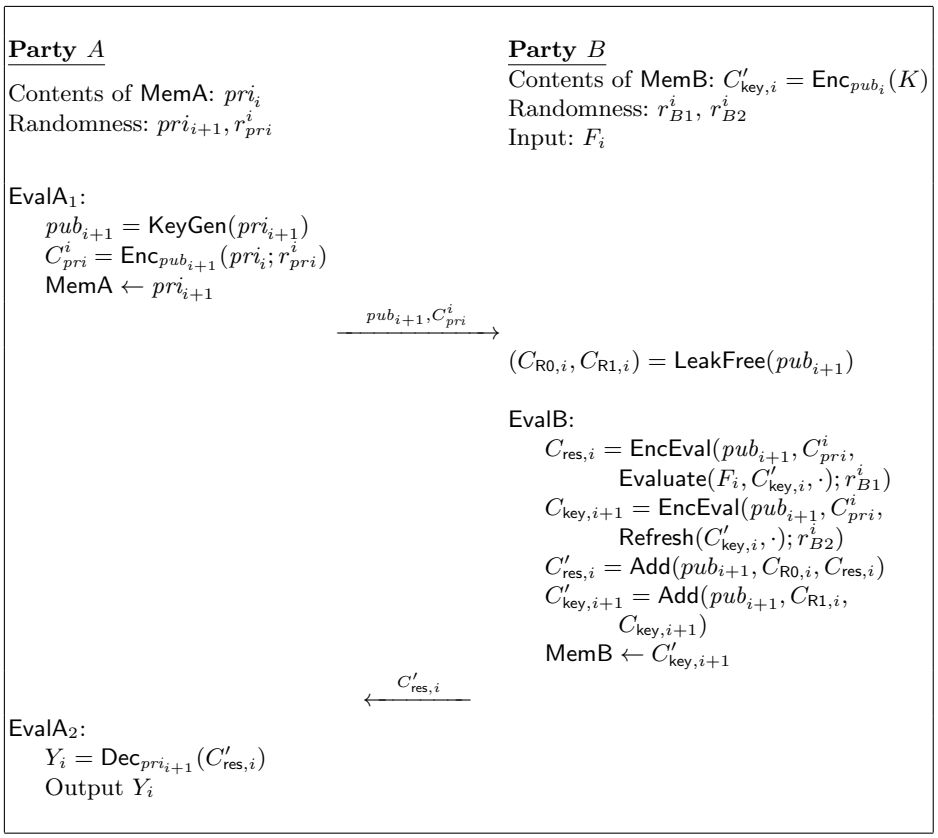

Fig. 2. The algorithm KPEval in its ith invocation.

The correctness of this construction follows in a straightforward manner from the correctness of the underlying FHE. We also note that our construction is *history free* according to Definition 6. This is due to the fact that the values assigned to MemA and MemB at the end of KPEval are independent from the function F. In particular, MemA is simply a random private key, and MemB contains an encryption of K which was obtained by a homomorphic evaluation

of Refresh on the previous contents of MemB and an encryption of the previous private key, neither of which depends on F.

The bulk of the analysis is in showing that our construction is in fact leakage-resilient according to Definition 3, where during each invocation the leakage structure on the computation of KPEval is given in Definition 5. We now state our main theorem.

Theorem 1. *Let* LRKP *be the 2-round split state key proxy described in the above construction, and let* $l : \mathbb{N} \to \mathbb{N}$. *If* FHE *is a* $2^{O(l(n))}$*-secure fully homomorphic encryption then* LRKP *is leakage-resilient against all* $O(l(n))$*-bounded adversaries in the OCL model.*

The theorem follows as a corollary from the following lemma:

Lemma 1. *Consider the experiment ExpReal instantiated using scheme* $LRKP$. *Then, for every function* $\varepsilon(n) > 0$, *every* $d > 0$, *every* $l : \mathbb{N} \to \mathbb{N}$, *and every* l*-bounded PPT adversary Adv that makes* n^d *queries and gets leakage according to the only-computation-leaks model, there exists a PPT simulator* S *such that if*

$$|\Pr[(Adv \leftrightarrows ExpReal) = 1] - \Pr[(Adv \leftrightarrows S) = 1]| \geq \varepsilon(n)$$

for infinitely many n, *then for every function* $\varepsilon'(n) > 0$ *there exists an adversary* Adv' *that runs in time*

$$\frac{2^{3l(n)+7}}{\varepsilon'(n)^2}\left(3l(n) + 4 + \log\frac{1}{\varepsilon'(n)}\right) \cdot time_n\left(LRKP \leftrightarrow Adv\right)$$

and breaks the semantic security of (KeyGen, Enc, Dec) *with advantage*

$$\frac{\varepsilon(n)}{3 \cdot 2^{2l(n)}(n^d + 1)} - 2\varepsilon'(n)$$

for infinitely many n. *Specifically,* S *runs in time* $time_n(LRKP \leftrightarrow Adv)$.

4.1 Proof approach for Lemma 1

Let Adv be a PPT adversary according to Definition 3 that makes n^d function evaluation queries and gets leakage according to the only-computation-leaks model described in Definition 5. We define a sequence of experiments where the initial experiment is the real security experiment ExpReal, and the final experiment is such that the leakage obtained by the adversary for each KPEval query F can be simulated given only $(F, F(K))$. Specifically, the final experiment involves instantiating our construction with key $\bar{0}$ instead of K. We show that if Adv can distinguish the initial experiment and the final experiment, we can construct an adversary Adv' that, roughly speaking, distinguishes variants of these experiments that consist of only two rounds. We then show how pairs of the leakage queries of Adv' can be combined into a single query (of larger output length) using a guess-and-check approach: when the adversary would normally

make the first of the pair of leakage queries, it instead guesses an output and verifies this guess when it makes the second leakage query; when the guess is wrong, the adversary outputs a randomly chosen bit. Repeatedly combining queries in this manner yields an adversary that just makes a single leakage query and (essentially) distinguishes encryptions of K and $\bar{0}$. To finish the proof, we use an observation of Akavia *et al* [1] that every $2^{O(\ell(n))}$-semantically-secure public-key encryption scheme remains secure when the adversary gets $O(\ell(n))$ bits of leakage on KeyGen. We defer the details to the full version of this paper [18].

5 Extensions and Applications

Below we describe some variants and applications of our scheme.

Resilience against simultaneous leakage. In Definition 5, the adversary is only allowed to see leakage from the part of memory where computation is occurring. Our construction is also secure under an alternative leakage model where the adversary is allowed to see *independent* leakage from *both* parts of memory each time it makes a leakage query. The basic idea is to first show that our construction is secure under a variant of Definition 5 where the adversary sees an additional leakage f_4 on memory B. Under this variant of Definition 5, the adversary's leakage queries strictly alternate between memory A and memory B. We then use an observation of Pietrzak [21] that simultaneous but independent leakage on two pieces of memory can be perfectly simulated by strictly alternating leakage (of twice the output length) on these two pieces of memory.

Resilience against complete compromise. Our scheme can be viewed as a protocol between two devices that communicate over a public channel. The key remains hidden even if the memory contents of one of the devices are leaked completely (for example, in a cold boot attack), provided that the compromise is detected and no further computation is performed using the counterpart device.

One-time programs. Our construction can be modified to work without any leak-free components by pre-computing a large number of tuples of the form (pri, pub, C, C') where C and C' are encryptions of 0 under pub, and storing the tuples in memory. Then, at each invocation, one such tuple is used (first pri and pub are used by EvalA_1, and then C, C' are used by EvalB). Assuming that only computation leaks information, the remaining tuples remain hidden until they are accessed. Therefore, security is obtained following essentially the same argument as the proof of Theorem 1. The number of invocations in this case is bounded by the number of pre-computed tuples. This approach provides a weaker security guarantee than the one time programs of [15] (i.e. only security against leakage), but has the advantage that the pre-computing phase is independent from the functionality that is being protected.

Concurrent composition. We have shown that an adversary interacting with a single instance of our construction gains no information about the underlying key. However, for some applications, such as private-key encryption where several

parties compute on the same agreed upon key, this may not suffice. It is quite possible that the adversary is performing side-channel attacks on several parties simultaneously, and is coordinating his leakage functions adaptively. In the full version of this paper, we show that an adversary interacting concurrently with several instances of our construction still gains no information through leakage.

Leakage-resilient private-key encryption. Extending the traditional notions of semantically secure encryption to the leakage setting is non-trivial. In particular, suppose that every invocation of the encryption algorithm leaks information. Then, since the plaintext of the adversary's challenge message is an input to the encryption algorithm, the adversary can trivially break semantic security by leaking even a single bit about this message. To deal with this problem, several works [9, 22, 8, 5] adopt the approach that the computation of the encryption of the challenge is *not* allowed to leak. We follow this approach, and show how to obtain semantically-secure private-key encryption in the leakage setting using LRKPs. The details are deferred to the full version of this paper.

Leakage-resilient public-key encryption. Constructions of public-key encryption schemes that are resilient to an a-priori bounded amount of leakage were recently given by [22, 2, 5]. However, no constructions are known of PKEs that remain secure under Chosen Ciphertext Attack (CCA), if the adversary can obtain leakage during each decryption query. LRKPs provide a convenient way to achieve such a construction. Specifically, given a CCA-PKE $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$, we construct a new PKE $(\mathsf{KeyGen}', \mathsf{Enc}, \mathsf{Dec}')$ where the encryption algorithm stays the same; the key generation KeyGen' runs KeyGen to obtain (pub, pri) and then initializes an LRKP with pri. The public key is pub, and the private key is the initial state state_1 of the LRKP. The decryption algorithm is stateful, and to decrypt a ciphertext C, Dec' generates a circuit $H(x)$ that computes that function $\mathsf{Dec}_x(C)$, and then uses KPEval to evaluate it on the private key pri.

Acknowledgements. We thank Charles Rackoff for many hours of discussion.

References

1. Akavia, A., Goldwasser, S., Vaikuntanathan, V.: Simultaneous hardcore bits and cryptography against memory attacks. In: TCC '09: Proceedings of the 6th Theory of Cryptography Conference. pp. 474–495. Springer, Berlin, Heidelberg (2009)
2. Alwen, J., Dodis, Y., Wichs, D.: Leakage resilient public-key cryptography in the bounded retrieval model. In: Advances in Cryptology — CRYPTO 2009. pp. 36–54. Springer, Berlin, Heidelberg (2009)
3. Brakerski, Z., Kalai, Y., Katz, J., Vaikuntanathan, V.: Cryptography resilient to continual memory leakage (2010), manuscript
4. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: EUROCRYPT (2010 (to appear))
5. Dodis, Y., Goldwasser, S., Kalai, Y., Peikert, C., Vaikuntanathan, V.: Public-key encryption schemes with auxiliary inputs (2009)
6. Dodis, Y., Goldwasser, S., Kalai, Y.T., Peikert, C., Vaikuntanathan, V.: Public-key encryption schemes with auxiliary inputs. In: TCC. pp. 361–381 (2010)

7. Dodis, Y., Haralambiev, K., Lopez-Alt, A., Wichs, D.: Cryptography against continuous memory attacks. Cryptology ePrint Archive, Report 2010/196 (2010), http://eprint.iacr.org/
8. Dodis, Y., Kalai, Y.T., Lovett, S.: On cryptography with auxiliary input. In: STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing. pp. 621–630. ACM, New York, NY, USA (2009)
9. Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: FOCS '08: Proceedings of the Annual IEEE Symposium on Foundations of Computer Science. pp. 293–302. IEEE Computer Society, Washington, DC, USA (2008)
10. Faust, S., Kiltz, E., Pietrzak, K., Rothblum, G.N.: Leakage-resilient signatures. In: TCC. pp. 343–360 (2010)
11. Faust, S., Rabin, T., Reyzin, L., Tromer, E., Vaikuntanathan, V.: Protecting against computationally bounded and noisy leakage. In: EUROCRYPT (2010 (to appear))
12. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing. pp. 169–178. ACM, New York, NY, USA (2009)
13. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM 43(3), 431–473 (1996)
14. Goldwasser, S., Kalai, Y., Peikert, C., Vaikuntanathan, V.: Robustness of the learning with errors assumption. In: Proceedings of the 1st Innovations in Computer Science conference (ICS 2010) (2010)
15. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: One-time programs. In: Advances in Cryptology — CRYPTO 2008. pp. 39–56. Springer, Berlin, Heidelberg (2008)
16. Goldwasser, S., Rothblum, G.: Securing computation against continuous leakage. These proceedings (2010)
17. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Advances in Cryptology — CRYPTO 2003. pp. 463–481. Springer, Berlin, Heidelberg (2003)
18. Juma, A., Vahlis, Y.: Protecting cryptographic keys against continual leakage. Cryptology ePrint Archive, Report 2010/205 (2010), http://eprint.iacr.org/
19. Katz, J., Vaikuntanathan, V.: Signature schemes with bounded leakage resilience. In: Matsui, M. (ed.) Advances in Cryptology - ASIACRYPT 2009. Proceedings. Lecture Notes in Computer Science, vol. 5912, pp. 703–720. Springer (2009), http://dx.doi.org/10.1007/978-3-642-10366-7
20. Micali, S., Reyzin, L.: Physically observable cryptography. In: TCC '04: Proceedings of the 1st Theory of Cryptography Conference. pp. 278–296. Springer, Berlin, Heidelberg (2004)
21. Pietrzak, K.: A leakage-resilient mode of operation. In: Advances in Cryptology – EUROCRYPT 2009. pp. 462–482. Springer-Verlag, Berlin, Heidelberg (2009)
22. Segev, G., Naor, M.: Public-key cryptosystems resilient to key leakage. In: Advances in Cryptology — CRYPTO 2009. pp. 18–35. Springer, Berlin, Heidelberg (2009)
23. Standaert, F.X., Malkin, T., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: Advances in Cryptology – EUROCRYPT 2009. pp. 443–461. Springer, Berlin, Heidelberg (2009)
24. Standaert, F.X., Pereira, O., Yu, Y., Quisquater, J.J., Yung, M., Oswald, E.: Leakage resilient cryptography in practice. Cryptology ePrint Archive, Report 2009/341 (2009), http://eprint.iacr.org/