# Unconditional UC-Secure Computation with (Stronger-Malicious) PUFs

Saikrishna Badrinarayanan[1], Dakshita Khurana[1(✉)], Rafail Ostrovsky[1], and Ivan Visconti[2]

[1] Department of Computer Science, UCLA, Los Angeles, USA
{saikrishna,dakshita,rafail}@cs.ucla.edu
[2] DIEM, University of Salerno, Salerno, Italy
visconti@unisa.it

**Abstract.** Brzuska et. al. (Crypto 2011) proved that unconditional UC-secure computation is possible if parties have access to honestly generated physically unclonable functions (PUFs). Dachman-Soled et. al. (Crypto 2014) then showed how to obtain unconditional UC secure computation based on malicious PUFs, assuming such PUFs are stateless. They also showed that unconditional oblivious transfer is impossible against an adversary that creates malicious stateful PUFs.

- In this work, we go beyond this seemingly tight result, by allowing any adversary to create stateful PUFs with a-priori bounded state. This relaxes the restriction on the power of the adversary (limited to stateless PUFs in previous feasibility results), therefore achieving improved security guarantees. This is also motivated by practical scenarios, where the size of a physical object may be used to compute an upper bound on the size of its memory.
- As a second contribution, we introduce a new model where any adversary is allowed to generate a malicious PUF that may encapsulate other (honestly generated) PUFs within it, such that the outer PUF has oracle access to all the inner PUFs. This is again a natural scenario, and in fact, similar adversaries have been studied in the tamper-proof hardware-token model (e.g., Chandran et. al. (Eurocrypt 2008)), but no such notion has ever been considered with respect to PUFs. All previous constructions of UC secure protocols suffer from explicit attacks in this stronger model.

In a direct improvement over previous results, we construct *UC protocols with unconditional security* in both these models.

---

The full version of the paper can be found at http://eprint.iacr.org/2016/636.

R. Ostrovsky—Research supported in part by NSF grant 1619348, US-Israel BSF grant 2012366, by DARPA Safeware program, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, and Lockheed-Martin Corporation Research Award. The views expressed are those of the authors and do not reflect position of the Department of Defense or the U.S. Government.

I. Visconti—Work done in part while visiting UCLA. Research supported in part by "GNCS - INdAM" and EU COST Action IC1306.


J.-S. Coron and J.B. Nielsen (Eds.): EUROCRYPT 2017, Part I, LNCS 10210, pp. 382–411, 2017.
DOI: 10.1007/978-3-319-56620-7_14

# 1 Introduction

In recent years, there has been a rich line of work studying how to enhance the computational capabilities of probabilistic polynomial-time players by making assumptions on hardware [33]. Two types of hardware assumptions in particular have had tremendous impact on recent research: tamper-proof hardware tokens and physically unclonable functions (PUFs).

The tamper-proof hardware token model introduced by Katz [24] relies on the simple and well accepted assumption that it is possible to physically protect a computing machine so that it can only be accessed as a black box, via oracle calls (as an example, think of smart cards). Immediately after its introduction, this model has been studied and its power is now understood in large part. Tamper-proof hardware tokens allow to obtain strong security notions and very efficient constructions, in some cases without requiring computational assumptions. In particular, the even more challenging case of stateless tokens started by [6] has been investigated further in [1,10,11,14,15,19,22,23,26].

## 1.1 Physically Unclonable Functions

Physically Unclonable Functions (PUFs) were introduced by Pappu et al. [28,29] but their actual potential has been understood only in recent years[1]. Increasing excitement over such *physical random oracles* generated various different (and sometimes incompatible) interpretations about the actual features and formalizations of PUFs.

Very roughly, a PUF is an object that can be queried by translating an input into a specific physical stimulation, and then by translating the physical effects of the stimulation to an output through a measurement. The primary appealing properties of PUFs include: (1) constructing two PUFs with similar input-output behavior is believed to be impossible (i.e. unclonability), and (2) the output of a PUF on a given input is seemingly unpredictable, i.e., one cannot "learn" the behavior of an honestly-generated PUF on any specific input without actually querying the PUF on that input.

There is a lot of ongoing exciting research on concrete constructions of PUFs, based on various technologies. As such, a PUF can only be described in an abstract way with the attempt to establish some target properties for PUF designers.

However, while formally modeling a PUF, one might (incorrectly) assume that a PUF guarantees some properties that unfortunately exceed the state of affairs in real-world scenarios. For example, assuming that the output of a genuine PUF is purely random is clearly excessive, while relying on min-entropy is certainly a safer and more conservative assumption. Various papers have proposed different models and even attempts to unify them. The interested reader

[1] PUFs are used in several applications like secure storage, RFID systems, anti-counterfeiting mechanisms, identification and authentication protocols [13,16,25,31,32,35].

can refer to [2] for detailed discussions about PUF models and their connections to properties of actual PUFs. We stress that in this work we will consider the use of PUFs in the UC model of [5]. Informally, this means that we want to study protocols that can securely compose with other protocols that may be executing concurrently.

### 1.2 UC Security Based on Physically Unclonable Functions

Starting with the work of Brzuska et al. [4], a series of papers have explored UC-secure computation based on physically unclonable functions. The goal of this line of cryptographic research has been to build protocols secure in progressively stronger models.

*The Trusted PUFs of Brzuska et al.* [4]. Brzuska et al. [4] began the first general attempts to add PUFs to the simulation paradigm of secure computation. They allowed any player (malicious or honest) to create only well-formed PUFs. As already mentioned, the output of a well-formed PUF on any arbitrary input is typically assumed to have sufficient min-entropy. Furthermore, on being queried with the same input, a well-formed PUF can be assumed to always produce identical (or sufficiently close) outputs. Applying error-tolerant fuzzy extractors [9] to the output ensures that each invocation of the PUF generates a (non-programmable) random string that can be reproduced by querying the PUF again with the same input. Brzuska et al. demonstrated how to obtain *unconditional* UC secure computation for any functionality in this model.

*The Malicious PUFs of Ostrovsky et al.* [27]. Ostrovsky et al. [27] then showed that the constructions of [4] become insecure in case the adversary can produce a *malicious* PUF that deviates from the behavior of an honest PUF. For instance, a malicious PUF could produce outputs according to a pseudo-random function rather than relying on physical phenomena, or it could just refuse to answer to a query. They also showed that it is possible to UC-securely compute any functionality using (potentially malicious) PUFs if one is willing to additionally make computational assumptions. They left open the problem of achieving *unconditional* UC-secure computation for any functionality using malicious PUFs.

Damgård and Scafuro [8] showed that *unconditional* UC secure commitments can be obtained even in the presence of malicious PUFs[2].

*The Fully Malicious but Stateless PUFs of Dachman-Soled et al.* [7]. More recently, it was shown by Dachman-Soled et al. [7] that unconditional UC security for general functionalities is impossible if the adversary is allowed to create malicious PUFs that can maintain state. They also gave a complementary feasibility result in an intermediate model where PUFs are allowed to be malicious, but are required to be stateless.

[2] This can be extended to other functionalities but not to all functionalities.

*We note that the impossibility result of* [7] *crucially relies on (malicious) PUFs being able to maintain a priori unbounded state.*

Thus, the impossibility seems interesting theoretically, but its impact to practical scenarios is unclear. In the real world, this result implies that unconditional UC secure computation of all functionalities is impossible in a model where an honest player is unable to distinguish maliciously created PUFs *with gigantic memory*, from honest (and therefore completely stateless) PUFs. One could argue that this allows the power of the adversary to go beyond the reach of current technology. On the other hand, the protocol of [7] breaks down completely if the adversary can generate a maliciously created PUF with even one bit of memory, and pass it off as a stateless (honest) PUF. This gap forms the starting point for our work.

### 1.3 Our Contributions

The current state-of-the-art leaves open the following question:

*Can we achieve UC-secure computation with malicious PUFs that are allowed to have a priori bounded state?*

In the main contribution of this work we answer this question in the affirmative. We show that not only it is possible to obtain UC-secure computation for any functionality as proven in [27] with computational assumptions, but we prove that this can be done with unconditional security, without relying on any computational assumptions. This brings us to our first main result, which we now state informally.

**Informal Theorem 1.** *For any two party functionality $\mathcal{F}$, there exists a protocol $\pi$ that unconditionally and UC-securely realizes $\mathcal{F}$ in the malicious bounded-stateful PUF model.*

As our second contribution, we introduce a new adversarial model for PUF-based protocols. Here, in addition to allowing the adversary to generate malicious stateless PUFs, we also allow him to encapsulate other (honestly generated) PUFs inside his own (malicious, stateless) PUF, even without the knowledge of the functionality of the inner PUFs. This allows the outer malicious PUF to make black-box (or oracle) calls to the inner PUFs that it encapsulates. In particular, the outer malicious PUF could answer honest queries by first making oracle calls to its inner PUFs, and generating its own output as a function of the output of the inner PUFs on these queries. An honest party interacting with such a malicious PUF need not be able to tell whether the PUF is malicious and possibly encapsulates other PUFs in it, or it is honest.

In this new adversarial model[3], we require all PUFs to be stateless. We will refer to this as the malicious encapsulated PUF model. It is interesting to

[3] A concurrent and independent work [30] considers an adversary that can encapsulate PUFs but does not propose UC-secure definitions/constructions.

note that all previously known protocols (even for limited functionalities such as commitments) suffer explicit attacks in this stronger malicious encapsulated (stateless) PUF model.

As our other main result, we develop techniques to obtain unconditional UC-secure computation in the malicious encapsulated PUF model.

**Informal Theorem 2.** *For any two party functionality $\mathcal{F}$, there exists a protocol $\pi$ that unconditionally and UC-securely realizes $\mathcal{F}$ in the malicious encapsulated (stateless) PUF model.*

Table 1 compares our results with prior work. Our feasibility result in the malicious bound-stateful PUF model and our feasibility result in the malicious encapsulated-stateless PUF model directly improve the works of [4,7]. Indeed each of our two results strengthen the power of the adversaries of [4,7] in one meaningful and natural direction still achieving the same unconditional results of [4,7]. A natural question is whether our techniques defeating malicious bounded-stateful PUFs can be composed with our techniques defeating malicious encapsulated-stateless PUFs to obtain unconditional UC-security for any functionality against adversaries that can construct malicious bounded-stateful encapsulated PUFs. While we do not see a priori any conceptual obstacle in obtaining such even stronger feasibility result, the resulting construction would be extremely complex and heavily tedious to analyze. Therefore we defer such a stronger claim to future work hoping that follow up research will achieve a more direct and elegant construction.

**Table 1.** The symbol ✓ (resp. ×) indicates that the construction satisfies (resp. does not satisfy) the corresponding security guarantee.

| Reference | Unconditional UC for any functionality | UC with stateless mal. PUFs | UC with bounded stateful mal. PUFs | UC with encapsulated stateless mal. PUFs |
|---|---|---|---|---|
| [4] | ✓ | × | × | × |
| [27] | × | ✓ | ✓ | × |
| [7] | ✓ | ✓ | × | × |
| This work | ✓ | ✓ | ✓ | × |
| This work | ✓ | ✓ | × | ✓ |

### 1.4 Our Techniques

The starting point for our constructions is the UC-secure OT protocol of [7], which itself builds upon the works of [4,27]. We begin by giving a simplified description of the construction in [7].

Suppose a sender $\mathcal{S}$ with inputs $(m_0, m_1)$ and a receiver $R$ with input bit $b$ want to run a UC secure OT protocol in the malicious stateless PUF model. Then, $\mathcal{S}$ generates a PUF and sends it to the receiver. The receiver queries the

PUF on a random challenge string $c$, records the output $r$ and then returns the PUF to $\mathcal{S}$. Then, the sender sends two random strings $(x_0, x_1)$ to the receiver. In turn, the receiver picks $x_b$, and sends $v = c \oplus x_b$ to the sender. The sender uses $\mathsf{PUF}(v \oplus x_0)$ to mask his input $m_0$ and $\mathsf{PUF}(v \oplus x_1)$, to mask his input $m_1$; and sends both masked values to the receiver. Here $\mathsf{PUF}(\cdot)$ denotes the output of the PUF on the given input. Since $\mathcal{R}$ had to return the PUF before $(x_0, x_1)$ were revealed, with overwhelming probability, $\mathcal{R}$ only knows $r = \mathsf{PUF}(v \oplus x_b)$, and can output one and only one of the masked sender inputs.

**Enhancing [7] in the Stateless PUF Model.** Though this was a simplified overview of the protocol in [7], it helps us to explain a subtle assumption required in their simulation strategy against a malicious sender. In particular, the simulator against a malicious sender must return the PUF to the sender before the sender picks random messages $(x_0, x_1)$. However, it is evident that in order to extract both messages $(m_0, m_1)$, the simulator must know $(x_0 \oplus x_1)$, and in particular know the response of the PUF on challenges $(c, c \oplus x_0 \oplus x_1)$ for some known string $c$.

But the simulator only learns $(x_0, x_1)$ after sending the PUF back to $\mathcal{S}$. Thus, in order to successfully extract the input of $\mathcal{S}$, the simulator should have the ability to make these queries *even after* the PUF has been returned to the malicious sender. This means that the PUF is supposed to remain accessible and untouched even when it is again in the hands of its malicious creator. We believe this is a very strong assumption that clearly deviates from real scenarios where the state of a PUF can easily be changed (e.g., by damaging it).

Our protocol in Fig. 1 gets rid of this strong assumption on the simulator, and we give a new sender simulation strategy that does not need to query the PUF when it is back in the hands of the malicious sender $\mathcal{S}$. This is also a first step in obtaining security against bounded-stateful PUFs. In the protocol of [7], if the PUF created by a malicious $\mathcal{S}$ is stateful, $\mathcal{S}$ on receiving the PUF can *first* change the state of the PUF (say, to output $\perp$ everywhere), and then output values $(x_0, x_1)$. In this case, no simulation strategy will be able to extract the inputs of the sender.

We change the protocol in [7], by having $\mathcal{S}$ commit to the random values $(x_0, x_1)$ at the beginning of the protocol, using a UC-secure commitment scheme. These values are decommitted only after $\mathcal{R}$ returns the PUF back to $\mathcal{S}$, so the scheme still remains UC-secure against a malicious receiver. Moreover, now the simulator against a malicious sender can use the straight-line extractor guaranteed by the UC-secure commitment scheme, to extract values $(x_0, x_1)$, and query the PUF on challenges of the form $(c, c \oplus x_0 \oplus x_1)$ for some string $c$. It then sets $v = c \oplus x_0$ and sends it to $\mathcal{S}$. Now, the sender masks are $\mathsf{PUF}(v \oplus x_0)$ and $\mathsf{PUF}(v \oplus x_1)$, which is nothing but $\mathsf{PUF}(c)$ and $\mathsf{PUF}(c \oplus x_0 \oplus x_1)$, which was already known to the sender simulator before returning the PUF to $\mathcal{S}$. This simulation strategy works (with the simulator requiring only black-box access to the malicious PUF's code) even if the PUF is later broken or its state is reset in any way. This protocol is described formally and proven secure in Sect. 3.

**Inputs:** Sender $\mathcal{S}$ has private inputs $(\mathsf{m}_0, \mathsf{m}_1) \in \{0,1\}^{2n}$ and Receiver $\mathcal{R}$ has private input $\mathsf{b} \in \{0,1\}$.

1. **Sender Message:** $\mathcal{S}$ does the following:
   - Generate a PUF $\mathsf{PUF}_s : \{0,1\}^n \to \{0,1\}^n$.
   - Choose a pair of random strings $(\mathsf{x}_0, \mathsf{x}_1) \xleftarrow{\$} \{0,1\}^{2n}$.
   - Send $\mathsf{PUF}_s$ and $(\mathsf{t}_0, \mathsf{t}_1) = \mathsf{UC\text{-}Com.Commit}(\mathsf{x}_0, \mathsf{x}_1)$ to $\mathcal{R}$.
2. **Receiver Message:** $\mathcal{R}$ does the following:
   - Choose a pair of random strings $(\mathsf{c}_0, \mathsf{c}_1) \xleftarrow{\$} \{0,1\}^{2n}$.
   - Compute $\mathsf{r}_0 = \mathsf{PUF}_s(\mathsf{c}_0), \mathsf{r}_1 = \mathsf{PUF}_s(\mathsf{c}_1)$.
   - Set $\mathsf{c} = \mathsf{c}_\mathsf{p}$ and $\mathsf{r} = \mathsf{r}_\mathsf{p}$ for $\mathsf{p} \xleftarrow{\$} \{0,1\}$.
   - Store the pair $(\mathsf{c}, \mathsf{r})$ and send $\mathsf{PUF}_s$ to $\mathcal{S}$.
3. **Sender Message:**
   - $\mathcal{S}$ sends $(\mathsf{x}_0, \mathsf{x}_1) = \mathsf{UC\text{-}Com.Decommit}(\mathsf{t}_0, \mathsf{t}_1)$ to $\mathcal{R}$.
4. **Receiver Message:** $\mathcal{R}$ does the following:
   - Abort if the decommitment does not verify correctly.
   - Compute and send $\mathsf{val} = \mathsf{c} \oplus \mathsf{x}_\mathsf{b}$ to $\mathcal{S}$.
5. **Sender Message:**
   - $\mathcal{S}$ computes $\mathsf{S}_0 = \mathsf{m}_0 \oplus \mathsf{PUF}_s(\mathsf{val} \oplus \mathsf{x}_0)$, $\mathsf{S}_1 = \mathsf{m}_1 \oplus \mathsf{PUF}_s(\mathsf{val} \oplus \mathsf{x}_1)$ and sends $(\mathsf{S}_0, \mathsf{S}_1)$ to $\mathcal{R}$.

**Outputs:** $\mathcal{S}$ has no output. $\mathcal{R}$ outputs $\mathsf{m}_\mathsf{b}$ which is computed as $(\mathsf{S}_\mathsf{b} \oplus \mathsf{r})$.

**Fig. 1.** Protocol $\Pi^1$ for 2-choose-1 OT in the malicious stateless PUF model.

**UC Security with Bounded Stateful PUFs.** A malicious PUF is allowed to maintain *state*, and can generate outputs (including $\perp$) as a function of not only the current query but also the previous queries that it received as input. This allows for some attacks on the protocol we just described, but they can be prevented by carefully interspersing coin-tossing with the protocol. Please see Sect. 4 for more details.

A stateful PUF created by the sender can also record information about the queries made by the receiver, and replay this information to a malicious sender when he inputs a secret challenge. Indeed, for PUFs with unbounded state, it is this ability to record queries that makes oblivious transfer impossible. However, we only consider PUFs that have a-priori bounded state. In this case, it is possible to design a protocol, parameterized by an upper bound on the size of the state of the PUF, that in effect exhausts the possible state space of such a malicious PUF. Our protocol then carefully uses this additional entropy to mask the inputs of the honest party.

More specifically, we repeat the OT protocol described before (with an additional coin-tossing phase) $K$ times in parallel, using the same (possibly malicious, stateful) $\mathsf{PUF}$, for sufficiently large $K > \ell$ (where $\ell$ denotes the upper bound on the state of the PUF). At this point, what we require essentially boils down to a *one-sided* malicious oblivious transfer extractor. This is a gadget that would

yield a single OT from $K$ leaky OTs, such that the single OT remains secure even when a malicious sender can ask for $\ell$ bits of universal leakage across all these OTs. This setting is incomparable to previously studied OT extractors [17,21] because: (a) we require a protocol that is secure against malicious (not just semi-honest) adversaries, and (b) the system has only one-sided leakage, i.e., a corrupt sender can request $\ell$ bits of leakage, but a corrupt receiver does not obtain any leakage at all.

For simplicity, we consider the setting of one-sided receiver leakage (instead of sender leakage). It is possible to consider this because OT is reversible. To protect against a malicious receiver that may obtain $\ell$ bits of universal leakage, the sender picks different random inputs for each OT execution, and then uses a strong randomness extractor to extract min-entropy and mask his inputs. We show that this in fact suffices to statistically hide the input messages of the sender. Please see Sect. 5 for a more detailed overview and construction.

**UC Security with Encapsulated PUFs.** We demonstrate the feasibility of UC secure computation, in a model where a party may (maliciously) encapsulate one or more PUFs that it obtained from honest parties, inside a malicious stateless PUF of its choice. We stress that our protocol itself does not require honest parties to encapsulate PUFs within each other.

To describe our techniques, we begin by revisiting the protocol in Fig. 1, that we described at the beginning of this overview. Suppose parties could maliciously encapsulate some honest PUFs inside a malicious PUF. Then a malicious receiver in this protocol, when it is supposed to return the sender's PUF $\mathsf{PUF}_s$, could instead return a *different* malicious PUF $\widehat{\mathsf{PUF}}_s$. In this case, the receiver would easily learn both inputs of the sender. But as correctly pointed out in prior work [7,8], the sender can deflect such attacks by probing and recording the output of $\mathsf{PUF}_s$ on some random input(s) (known as Test Queries) before sending it to the receiver. Later the sender can check whether $\widehat{\mathsf{PUF}}_s$ correctly answers to all Test Queries.

However, a malicious receiver may create $\widehat{\mathsf{PUF}}_s$ that encapsulates $\mathsf{PUF}_s$, such that $\widehat{\mathsf{PUF}}_s$ is programmed to send most outer queries to $\mathsf{PUF}_s$ and echo its output externally; in order to pass the sender's test. However, $\widehat{\mathsf{PUF}}_s$ may have its own malicious procedure to evaluate some of the other external queries. In particular, the "unpredictability" of $\widehat{\mathsf{PUF}}_s$ may break down completely on these queries.

It turns out that the security of the sender in the basic OT protocol of Fig. 1 hinges on the unpredictability of the output of $\mathsf{PUF}_s$ (in this situation, $\widehat{\mathsf{PUF}}_s$) on a "special challenge query" only, which we will denote by $\mathfrak{s}$. It is completely feasible for a receiver to create a malicious encapsulating PUF $\widehat{\mathsf{PUF}}_s$ that passes the sender tests, and yet its output on this special query $\mathfrak{s}$ is completely known to the receiver, therefore breaking sender security.

We overcome this issue by ensuring that $\mathfrak{s}$ is chosen using a coin toss, and is completely unknown to the receiver until after he has sent $\widehat{\mathsf{PUF}}_s$ (possibly a malicious encapsulating PUF) back to the sender. Intuitively, this means that $\widehat{\mathsf{PUF}}_s$ will either not pass the sender tests, or will be highly likely to deflect this

the query $\mathfrak{s}$ to the inner PUF and echo its output (thereby ensuring that the output of the PUF on input $\mathfrak{s}$ is unpredictable for the receiver). An additional subtlety that arises is that the receiver might use an incorrect $\mathfrak{s}$ in the protocol (instead of using the output of the coin toss): the receiver is forced to use the correct $\mathfrak{s}$ via a special cut-and-choose mechanism. For a more detailed overview and construction, please see to Sect. 6.

**UC-Secure Commitments Against Encapsulation Attacks.** Finally, UC-secure commitments against encapsulation attacks play a crucial role in our UC-secure OT protocol in the encapsulation model. But, we note that the basic commitment protocol of [8] is insecure in this stronger model, and therefore we modify the protocol of [8] to achieve UC-security in this scenario. In a nutshell, this is done by having the receiver send an additional PUF at the end of the protocol, and forcing any malicious committer to query this additional PUF on the committer's input bit. We then show that even an encapsulating (malicious) committer will have to carry out this step honestly in order to complete the commit phase. Then, a simulator can extract the adversary's committed value by observing the queries of the malicious committer to this additional PUF. We illustrate in detail, how prior constructions of UC-secure commitments fail in the PUF encapsulation model in Sect. 7. Our UC-secure commitment protocol in the encapsulated malicious (stateless) PUF model is also described in Sect. 7.

### 1.5 Organization

The rest of this paper is organized as follows. In Sect. 2, we discuss PUFs and other preliminaries relevant to our protocols. In Sect. 3, we describe an improved version of the protocol in [7], in the stateless PUF model. In Sects. 4 and 5, we boost this protocol to obtain security in the bounded stateful PUF model. In Sects. 6 and 7, we discuss protocols that are secure in the PUF encapsulation model. In Appendix A, we discuss the formal modelling of our PUFs. The complete models and proofs that could not be included in this version owing to space restrictions, can be found in the full version of the paper.

## 2 Preliminaries

### 2.1 Physically Unclonable Functions

A PUF is a noisy physical source of randomness. The randomness property comes from an uncontrollable manufacturing process. A PUF is evaluated with a physical stimulus, called the *challenge*, and its physical output, called the *response*, is measured. Since the processes involved are physical, the function implemented by a PUF can not (necessarily) be modeled as a mathematical function, neither can be considered computable in PPT. Moreover, the output of a PUF is noisy, namely, querying a PUF twice with the same challenge, could yield distinct responses within a small Hamming distance to each other.

Moreover, the response need not be random-looking; rather, it is a string drawn from a distribution with high min-entropy. Prior work has shown that, using fuzzy extractors, one can eliminate the noisiness of the PUF and make its output uniformly random. For simplicity, we assume this in the body of the paper and give a detailed description in the full version.

A PUF-family is a pair of (not necessarily efficient) algorithms $\mathsf{Sample}$ and $\mathsf{Eval}$. Algorithm $\mathsf{Sample}$ abstracts the PUF fabrication process and works as follows. On input the security parameter, it outputs a PUF-index $\mathsf{id}$ from the PUF-family satisfying the security properties (that we define soon) according to the security parameter. Algorithm $\mathsf{Eval}$ abstracts the PUF-evaluation process. On input a challenge $q$, it evaluates the PUF on $q$ and outputs the response $a$ of length $rg$, denoting the range. Without loss of generality, we assume that the challenge space of a PUF is a full set of strings of a certain length.

*Security of PUFs.* Following [4], we consider only the two main security properties of PUFs: *unclonability* and *unpredictability*. Informally, unpredictability means that the output of the PUF is statistically indistinguishable from a uniform random string. Formally, unpredictability is modeled via an entropy condition on the PUF distribution. Namely, given that a PUF has been measured on a polynomial number of challenges, the response of the PUF evaluated on a new challenge still has a significant amount of entropy. For simplicity, a PUF is unpredictable if its output on any given input appears uniformly random.

Informally, unclonability states that in a protocol consisting of several parties, only the party in whose possession the PUF is, can evaluate the PUF. When a party sends a PUF to a different party, it can no longer evaluate the PUF till the time it gets the PUF back. Thus a party not in possession of a PUF cannot predict the output of the PUF on an input for which it did not query the PUF, unless it maliciously created the PUF. A formal definition of unclonability is given in the full version of this paper.

A PUF can be modeled as an ideal functionality $\mathcal{F}_{\mathsf{PUF}}$, which mimics the behavior of the PUF in the real world. We formally define ideal functionalities corresponding to honestly generated and various kinds of maliciously generated PUFs in Appendix A. We summarize these here: the model for honestly generated PUFs and for malicious stateless/stateful PUFs has been explored in prior work [7,27], and we introduce the model for encapsulated PUFs.

- **An honestly generated PUF** can be created according to a sampling algorithm $\mathsf{Samp}$, and evaluated honestly using an evaluation algorithm $\mathsf{Eval}$. The output of an honestly generated PUF is *unpredictable* even to the party that created it, i.e., even the creator cannot predict the output of an honestly generated PUF on any given input without querying the PUF on that input.
- **A malicious stateless PUF**, on the other hand, can be created by the adversary substituting an $\mathsf{Eval}_{mal}$ procedure of his choice for the honest $\mathsf{Eval}$ procedure. Whenever a (honest) party in possession of this PUF evaluates the PUF, it runs the stateless procedure $\mathsf{Eval}_{mal}(c)$ instead of $\mathsf{Eval}(c)$ (and cannot distinguish $\mathsf{Eval}_{mal}(c)$ from $\mathsf{Eval}(c)$ unless they are distinguishable with

black-box access to the PUF). The output of such a PUF cannot depend on previous queries, moreover no adversary that creates the PUF but does not possess it, can learn previous queries made to the PUF when it was not in its possession. We adapt the definitions from [27], where $\mathsf{Eval}_{mal}$ is a polynomial-time algorithm with oracle access to $\mathsf{Eval}$. This is done to model the fact that the $\mathsf{Eval}_{mal}$ algorithm can access an (honest) source of randomness $\mathsf{Eval}$, and can arbitrarily modify its output using any polynomial-time strategy.

– **A malicious stateful PUF** can be created by the adversary substituting a stateful $\mathsf{Eval}_{mal}$ procedure of his choice for the honest $\mathsf{Eval}$ procedure. Whenever a party in possession of this PUF evaluates the PUF, it runs the stateful procedure $\mathsf{Eval}_{mal}(c)$ instead of $\mathsf{Eval}(c)$. Thus, the output of a stateful malicious PUF can possibly depend on previous queries, moreover an adversary that created a PUF can learn previous queries made to the $\mathsf{PUF}$ by querying it, say, on a secret input. $\mathsf{Eval}_{mal}$ is a polynomial-time stateful Turing Machine with oracle access to $\mathsf{Eval}$. Again, this is done to model the fact that the $\mathsf{Eval}_{mal}$ algorithm can access an (honest) source of randomness, $\mathsf{Eval}$, and arbitrarily modify its output using any polynomial-time strategy. Malicious stateful PUFs can further be of two types:
  - *Bounded Stateful.* Such a PUF can maintain a-priori bounded memory/state (which it may rewrite, as long as the total memory is bounded).
  - *Unbounded Stateful.* Such a PUF can maintain unbounded memory/state.
– **A malicious encapsulating PUF** can possibly encapsulate other (honestly generated) PUFs inside it[4], without knowing the functionality of these inner PUFs. Such a PUF $\mathsf{PUF}_{\mathtt{mal}}$ can make black-box calls to the inner PUFs, and generate its outputs as a function of the output of the inner (honest) PUFs.
  This is modeled by having the adversary substitute an $\mathsf{Eval}_{mal}$ procedure of his choice for the honest $\mathsf{Eval}$ procedure in the $\mathsf{PUF}_{\mathtt{mal}}$ that it creates, where as usual $\mathsf{Eval}_{mal}$ is a polynomial-time Turing Machine with oracle access to $\mathsf{Eval}$. Similar to the two previous bullets, this is done to model the fact that the $\mathsf{Eval}_{mal}$ algorithm can access an (honest) source of randomness, $\mathsf{Eval}$, and arbitrarily modify its output using any polynomial-time strategy.
  In addition, $\mathsf{Eval}_{mal}$ can also make oracle calls to polynomially many other (honestly generated) procedures $\mathsf{Eval}_1, \mathsf{Eval}_2, \ldots \mathsf{Eval}_M$ that are contained in PUFs $\mathsf{PUF}_1, \mathsf{PUF}_2, \ldots \mathsf{PUF}_M$, for any a-priori unbounded $M = \mathsf{poly}(n)$. These correspond to honestly generated PUFs that the adversary may be encapsulating within its own malicious PUF. Thus on some input $c$, the $\mathsf{Eval}_{mal}$ procedure may make oracle calls to $\mathsf{Eval}_1, \mathsf{Eval}_2, \ldots \mathsf{Eval}_M$ on polynomially many inputs, and compute its output as a function of the outputs of the $\mathsf{Eval}, \mathsf{Eval}_1, \mathsf{Eval}_2, \ldots \mathsf{Eval}_M$ procedures. Of course, we ensure that the adversary's $\mathsf{Eval}_{mal}$ procedure can make calls to some honestly generated procedure $\mathsf{Eval}_i$ *only if* the adversary owns the PUF $\mathsf{PUF}_i$ implementing the $\mathsf{Eval}_i$ procedure when creating the encapsulating malicious PUF. Furthermore, when the

[4] Since the adversary knows the code of maliciously generated PUFs, this model automatically captures real-world scenarios where an adversary may be encapsulating other malicious PUFs inside its own.

adversary passes such a PUF to an honest party, the adversary "loses ownership" of $\mathsf{PUF}_i$ and is no longer allowed to access the $\mathsf{Eval}_i$ procedure, this is similar to the unclonability requirement. This is modeled by assigning an owner to each PUF, and on passing an outer (encapsulating) PUF to an honest party, the adversary must automatically pass all the inner (encapsulated) honest PUFs. Whenever an honest party is in possession of such an adversarial PUF $\mathsf{PUF}_{\mathtt{mal}}$ and evaluates it, it receives the output of $\mathsf{Eval}_{mal}$. When the adversary is allowed to construct encapsulating PUFS, we restrict all PUFs to be stateless. Therefore the model with encapsulating PUFs is incomparable with the model with bounded-stateful malicious PUFs. Further details on the modeling of malicious stateless PUFs that may encapsulate other stateless PUFs, are provided in Appendix A.

To simplify notation, we write $\mathsf{PUF} \leftarrow \mathsf{Sample}(1^K)$, $r = \mathsf{PUF}(c)$ and assume that $\mathsf{PUF}$ is a deterministic function with random output.

### 2.2 UC Secure Computation

The UC framework, introduced by [5] is a strong framework which gives security guarantees even when protocols may be arbitrarily composed.

*Commitments.* A UC-secure commitment scheme $\mathsf{UC\text{-}Com}$ consists of the usual commitment and decommitment algorithms, along with (straight-line) procedures allowing the simulator to extract the committed value of the adversary and to equivocate a value that the simulator committed to. We denote these by ($\mathsf{UC\text{-}Com.Commit}$, $\mathsf{UC\text{-}Com.Decommit}$, $\mathsf{UC\text{-}Com.Extract}$, $\mathsf{UC\text{-}Com.Equivocate}$). Damgård and Scafuro [8] realized unconditional UC secure commitments using stateless PUFs, in the malicious stateful PUF model.

*OT.* Ideal 2-choose-1 oblivious transfer (OT) is a two-party functionality that takes two inputs $m_0, m_1$ from a sender and a bit $b$ from a receiver. It outputs $m_b$ to the receiver and $\perp$ to the sender. We use $\mathcal{F}_{\mathsf{ot}}$ to denote this functionality. Given UC oblivious transfer, it is possible to obtain UC secure two-party computation of any functionality.

Formal definitions of these functionalities and background on prior results are provided in the full version of this paper.

## 3 Unconditional UC Security with (Malicious) Stateless PUFs

As a warm up, we start by considering malicious stateless PUFs as in [7] and we strengthen their protocol in order to achieve security even when the simulator does not have access to a malicious PUF that is in possession of the adversary that created it.

*Construction.* Let $n$ denote the security parameter. The protocol $\Pi^1$ in Fig. 2 UC-securely and unconditionally realizes 2-choose-1 OT in the malicious stateless PUF model, between a sender $\mathcal{S}$ and receiver $\mathcal{R}$, with the following restrictions:

1. The random variables $(x_0, x_1)$ are chosen by $\mathcal{S}$ independently of $\mathsf{PUF}_s$[5].
2. A (malicious) $\mathcal{R}$ returns to $\mathcal{S}$ the same PUF, $\mathsf{PUF}_s$ that it received[6].

We enforce these restrictions in this section only for simplicity and modularity purposes. We remove them in Sects. 4 and 6 respectively.

**Inputs:** Sender $\mathcal{S}$ has private inputs $(\mathsf{m}_0, \mathsf{m}_1) \in \{0,1\}^{2n}$ and Receiver $\mathcal{R}$ has private input $\mathsf{b} \in \{0,1\}$.

1. **Sender Message:** $\mathcal{S}$ does the following:
   - Generate a PUF $\mathsf{PUF}_s : \{0,1\}^n \to \{0,1\}^n$.
   - Choose a pair of random strings $(\mathsf{x}_0, \mathsf{x}_1) \xleftarrow{\$} \{0,1\}^{2n}$.
   - Send $\mathsf{PUF}_s$ and $(\mathsf{t}_0, \mathsf{t}_1) = \mathsf{UC\text{-}Com.Commit}(\mathsf{x}_0, \mathsf{x}_1)$ to $\mathcal{R}$.
2. **Receiver Message:** $\mathcal{R}$ does the following:
   - Choose a pair of random strings $(\mathsf{c}_0, \mathsf{c}_1) \xleftarrow{\$} \{0,1\}^{2n}$.
   - Compute $\mathsf{r}_0 = \mathsf{PUF}_s(\mathsf{c}_0), \mathsf{r}_1 = \mathsf{PUF}_s(\mathsf{c}_1)$.
   - Set $\mathsf{c} = \mathsf{c}_\mathsf{p}$ and $\mathsf{r} = \mathsf{r}_\mathsf{p}$ for $\mathsf{p} \xleftarrow{\$} \{0,1\}$.
   - Store the pair $(\mathsf{c}, \mathsf{r})$ and send $\mathsf{PUF}_s$ to $\mathcal{S}$.
3. **Sender Message:**
   - $\mathcal{S}$ sends $(\mathsf{x}_0, \mathsf{x}_1) = \mathsf{UC\text{-}Com.Decommit}(\mathsf{t}_0, \mathsf{t}_1)$ to $\mathcal{R}$.
4. **Receiver Message:** $\mathcal{R}$ does the following:
   - Abort if the decommitment does not verify correctly.
   - Compute and send $\mathsf{val} = \mathsf{c} \oplus \mathsf{x}_\mathsf{b}$ to $\mathcal{S}$.
5. **Sender Message:**
   - $\mathcal{S}$ computes $\mathsf{S}_0 = \mathsf{m}_0 \oplus \mathsf{PUF}_s(\mathsf{val} \oplus \mathsf{x}_0)$, $\mathsf{S}_1 = \mathsf{m}_1 \oplus \mathsf{PUF}_s(\mathsf{val} \oplus \mathsf{x}_1)$ and sends $(\mathsf{S}_0, \mathsf{S}_1)$ to $\mathcal{R}$.

**Outputs:** $\mathcal{S}$ has no output. $\mathcal{R}$ outputs $\mathsf{m}_\mathsf{b}$ which is computed as $(\mathsf{S}_\mathsf{b} \oplus \mathsf{r})$.

**Fig. 2.** Protocol $\Pi^1$ for 2-choose-1 OT in the malicious stateless PUF model.

Our protocol makes black-box use of a UC-commitment scheme, denoted by the algorithms $\mathsf{UC\text{-}Com.Commit}$ and $\mathsf{UC\text{-}Com.Decommit}$. We use $\mathsf{UC\text{-}Com.Commit}(a, b)$ to denote a commitment to the concatenation of strings $a$ and $b$.

[5] This is fixed later by using coin-tossing to generate $(x_0, x_1)$, see Sect. 4.

[6] In Sect. 6, we consider an even stronger model where $\mathcal{R}$ may encapsulate $\mathsf{PUF}_s$ within a possibly malicious $\widehat{\mathsf{PUF}}_s$. $\widehat{\mathsf{PUF}}_s$ externally forwards some queries to $\mathsf{PUF}_s$ and forwards the outputs to the evaluator, while possibly replacing some or all of these outputs with other arbitrary values. We note that this covers the case where the receiver generates $\widehat{\mathsf{PUF}}_s$ malicious and independently of $\mathsf{PUF}_s$.

UC-secure commitments can be unconditionally realized in the malicious stateless PUF model [8]. Formally, we prove the following theorem:

**Theorem 1.** *The protocol $\Pi^1$ in Fig. 2 unconditionally UC-securely realizes $\mathcal{F}_{\mathsf{ot}}$ in the malicious stateless PUF model.*

This protocol is essentially the protocol of Dachman-Soled et al. [7], modified to enable correct extraction of the sender's input. The protocol as specified in [7], even though private, does not allow for straight-line extraction of the sender's input messages, unless one is willing to make the strong assumption that the simulator can make queries to a (malicious) PUF that an adversary created, even when this malicious PUF is in the adversary's possession (i.e., the adversary is forced not to update nor to damage/destroy the PUF).

Our main modification is to have the sender commit to his values $(x_0, x_1)$ using a UC-secure commitment scheme. In this case, it is possible for the simulator to extract $(x_0, x_1)$ in a straight-line manner from the commitment, and therefore extract the sender's input while it remains hidden from a real receiver. The rest of the proof follows in the same manner as [7]; recall that we already gave an overview in Sect. 1.4. The formal proofs of correctness and security can be found in the full version of this paper.

## 4 UC-Security with (Bounded-Stateful Malicious) PUFs

*Overview.* A malicious *stateful* PUF can generate outputs as a function of its previous input queries. For the (previous) protocol in Fig. 2, note that in Step 2, $\mathsf{Sim}_{\mathcal{S}}$ makes two queries $(c_1, c_2)$ to the PUF such that $(c_1 \oplus c_2) = (x_1 \oplus x_2)$, where $(x_1, x_2)$ are the sender's random messages. On the other hand, an honest receiver makes two queries $(c_1, c_2)$ to the PUF such that $(c_1 \oplus c_2) = \mathsf{rv}$, for an independent random variable $\mathsf{rv}$.

Therefore, when combined with the sender's view, the joint distribution of the evaluation queries made to the PUF by $\mathsf{Sim}_{\mathcal{S}}$, differs from the joint distribution of the evaluation queries made to the PUF by an honest receiver. Thus, a malicious sender can distinguish the two worlds by having a malicious stateful PUF compute a reply to $c_2$ depending on the value of the previous challenge $c_1$. We will call these attacks of Type I. In this section, we will describe a protocol secure against all possible attacks where a stateful PUF computes responses to future queries as a function of prior queries.

A stateful PUF created by the sender can also record information about the queries made by the receiver, and replay this information to a malicious sender when he inputs a secret challenge. For PUFs with bounded state, we view these as 'leakage' attacks, by considering all information recorded and replayed by a PUF as leakage. We will call these attacks of Type II. We describe a protocol secure against general bounded stateful PUFs (i.e., secure against attacks of both Type I and Type II) in Sect. 5.

Repeat the following protocol $K$ times in parallel for fresh private inputs $(\mathsf{m}_0^i, \mathsf{m}_1^i)$ of the sender and $\mathsf{b}_i$ of the receiver for $i \in [K]$.
**Inputs:** Sender $\mathcal{S}$ has private inputs $(\mathsf{m}_0, \mathsf{m}_1) = (\mathsf{m}_0^i, \mathsf{m}_1^i) \in \{0,1\}^{2n}$ and Receiver $\mathcal{R}$ has private input $\mathsf{b} = \mathsf{b}^i \in \{0,1\}$.

1. **Sender Message:** $\mathcal{S}$ does the following.
   - Generate a PUF $\mathsf{PUF}_s : \{0,1\}^n \to \{0,1\}^n$.(Use the same PUF for all the $K$ parallel sessions).
   - Choose a pair of random strings $(\mathsf{x}_0, \mathsf{x}_1) \xleftarrow{\$} \{0,1\}^{2n}$.
   - Send $\mathsf{PUF}_s$ and $(\mathsf{t}_0, \mathsf{t}_1) = \mathsf{UC\text{-}Com.Commit}(\mathsf{x}_0, \mathsf{x}_1)$ to $\mathcal{R}$.
2. **Receiver Message:** $\mathcal{R}$ does the following.
   - Choose a pair of random strings $(\mathsf{c}_0, \mathsf{c}_1) \xleftarrow{\$} \{0,1\}^{2n}$.
   - Compute $\mathsf{r}_0 = \mathsf{PUF}_s(\mathsf{c}_0), \mathsf{r}_1 = \mathsf{PUF}_s(\mathsf{c}_1)$.
   - Set $\mathsf{c} = \mathsf{c}_\mathsf{p}$ and $\mathsf{r} = \mathsf{r}_\mathsf{p}$ for $\mathsf{p} \xleftarrow{\$} \{0,1\}$ and store the pair $(\mathsf{c}, \mathsf{r})$.
   - Pick and send $(\hat{\mathsf{x}}_0, \hat{\mathsf{x}}_1) \xleftarrow{\$} \{0,1\}^{2n}$ along with $\mathsf{PUF}_s$, to $\mathcal{S}$.
3. **Sender Message:**
   $\mathcal{S}$ sends $(\mathsf{x}_0, \mathsf{x}_1) = \mathsf{UC\text{-}Com.Decommit}(\mathsf{t}_0, \mathsf{t}_1)$ to $\mathcal{R}$.
4. **Receiver Message:** If $\mathsf{UC\text{-}Com.Decommit}(\mathsf{t}_0, \mathsf{t}_1)$ does not verify, abort. Else, compute and send $\mathsf{val} = \mathsf{c} \oplus \mathsf{x}_\mathsf{b} \underline{\oplus \hat{\mathsf{x}}_\mathsf{b}}$ to $\mathcal{S}$.
5. **Sender Message:** $\mathcal{S}$ does the following.
   - Compute
     $\mathsf{S}_0 = \mathsf{m}_0 \oplus \mathsf{PUF}_s(\mathsf{val} \oplus \mathsf{x}_0 \underline{\oplus \hat{\mathsf{x}}_0})$ and $\mathsf{S}_1 = \mathsf{m}_1 \oplus \mathsf{PUF}_s(\mathsf{val} \oplus \mathsf{x}_1 \underline{\oplus \hat{\mathsf{x}}_1})$.
   - Send $(\mathsf{S}_0, \mathsf{S}_1)$ to $\mathcal{R}$.

**Outputs:** $\mathcal{S}$ has no output. $\mathcal{R}$ outputs $\mathsf{m}_\mathsf{b}$ which is computed as $(\mathsf{S}_\mathsf{b} \oplus \mathsf{r})$.

**Fig. 3.** Protocol $\Pi_K$ for $K$ 2-choose-1 OTs (with at most $\ell$-bounded leakage) in the malicious stateful PUF model. The changes from the protocol in Fig. 2 are underlined.

*Our Strategy.* Let $\ell$ denote a polynomial upper bound on the size of the memory of any malicious PUF created by the sender $\mathcal{S}$. Our strategy to obtain secure oblivious transfer from any PUF with $\ell$-bounded state is as follows: We use (the same) $\mathsf{PUF}_s$ created by the sender, to execute $K = \Theta(\ell)$ oblivious transfers in parallel. In our new protocol in Fig. 3, we carefully intersperse an additional round of coin tossing with our basic protocol from Fig. 2, to obtain security against attacks of Type I.

Specifically, we modify the protocol of Fig. 2 as follows: instead of having $\mathcal{S}$ generate the random strings $(x_0, x_1)$, we set the protocol up so that *both* $\mathcal{S}$ and the receiver $\mathcal{R}$ generate XOR shares of $(x_0, x_1)$. Furthermore, $\mathcal{R}$ generates his shares only after obtaining the PUF and a commitment to sender shares from $\mathcal{S}$. In such a case, the PUF created by $\mathcal{S}$ must necessarily be independent of the receiver shares and consequently, also independent of $(x_0, x_1)$.

Recall from Sect. 3, that the simulator against a malicious sender succeeds if it can obtain the output of the PUF to queries of the form $(c, c \oplus x_0 \oplus x_1)$ for a random $c$, whereas an honest receiver can only make queries of the form $(c_1, c_2)$ for randomly chosen $(c_1, c_2)$. Since $(x_0, x_1)$ appear to be distributed uniformly at random to the PUF, the distributions of $(c, c \oplus x_0 \oplus x_1)$ and $(c_1, c_2)$ are

also statistically indistinguishable to the PUF[7]. Therefore, the sender simulator succeeds whenever the honest receiver does not abort and this suffices to prove security against a malicious sender.

Finally, we note that the simulation strategy against a malicious receiver remains similar to one of Sect. 3, even if the receiver has the ability to create PUFs with unbounded state.

*Construction.* The protocol $\Pi_K$ in Fig. 3 allows us to use an $\ell$-bounded stateful PUF to obtain $K$ secure (but one-sided leaky) oblivious transfers, such that a malicious sender can obtain at most $\ell$ bits of additional universal leakage on the joint distribution of the receiver's choice input bits $(b_1, b_2, \ldots b_K)$. Our protocol makes black-box use of a UC-commitment scheme, denoted by the algorithms $\mathsf{UC\text{-}Com.Commit}$ and $\mathsf{UC\text{-}Com.Decommit}$[8]. UC-secure commitments can be unconditionally realized in the malicious *stateful* PUF model [8].

**Theorem 2.** *The protocol $\Pi_K$ unconditionally UC-securely realizes $K$ instances of $OT(\mathcal{F}_{\mathsf{ot}}^{[\otimes K]})$ in an $\ell$-bounded-stateful PUF model, except that a malicious sender can obtain at most $\ell$ bits of additional universal leakage on joint distribution of the receiver's choice bits over all $\mathcal{F}_{\mathsf{ot}}^{[\otimes K]}$.*

Correctness is immediate from inspection, and the complete proof of security is in the full version of the paper.

[7] We assume the simulator can control which simulator queries the adversary's PUF records (but an honest party cannot). Indeed, without our assumption, if a stateful PUF recorded every simulator query, a malicious sender on getting back $\mathsf{PUF}_s$ may observe the correlation between queries $(c, c')$ recorded by the PUF when the simulator queried it, versus two random queries when an actual honest party queried it. Ours is a natural assumption and obtaining secure OT remains extremely non-trivial even with this assumption. We note that this requirement can be removed using standard secret sharing along with cut-and-choose, but at the cost of a more complicated protocol with a worse OT production rate. This protocol is described in the full version of this paper.

[8] The UC framework (and its variants) seemingly fail to capture the possibility of transfer of physical devices like PUFs across different protocols, to the best of our knowledge. Within our OT protocol, we invoke the ideal functionality for UC-secure commitments. Thus, we would like to ensure that our UC-secure commitment scheme composes with the rest of the protocol even if PUFs created in the commitment scheme are used elsewhere in the OT protocol and vice versa. In our protocol, the only situation where such an issue might arise, is if one of the parties in the main OT protocol, later maliciously passes a PUF that it received from the honest party during a commitment phase. This is avoided by requiring all parties to return the PUFs to their original creator at the end of the decommitment phase. Note that this does not violate security even if the PUFs are malicious and stateful. The creating party, like in previous works [7,8] can probe a random point before sending the PUF, and then check this point again on receiving the PUF, to ensure that they received the correct PUF. Generic results attempting to model UC security in presence of physical devices that can be transferred across different protocol executions have been presented in [3,20].

## 5 One-Sided Correlation Extractors with Malicious Security

From Sect. 4, in the $\ell$-bounded stateful PUF model, we obtain $K$ leaky oblivious transfers, such that the sender can obtain $\ell$ bits of universal leakage on the joint distribution of the receiver's choice bits over all $K$ oblivious transfers.

Because OT is reversible [37], it suffices to consider a reversed version of the above setting, i.e., where the receiver can obtain $\ell$ bits of additional universal leakage on the joint distribution of all the sender's messages over all $K$ oblivious transfers. More formally, the leakage model we consider is as follows:

*One-Sided Leakage Model for Correlation Extractors.* Here, we begin by describing our leakage model for OT correlations formally, and then we define one-sided correlation extractors for OT. Our leakage model is as follows:

1. **$K$-OT Correlation Generation Phase:** For $i \in [K]$, the sender $\mathcal{S}$ obtains $(x_0^i, x_1^i) \in \{0,1\}^2$ and the receiver $\mathcal{R}$ gets $(b_i, x_{b_i}^i)$.
2. **Corruption and Leakage Phase:** A malicious adversary corrupts the receiver and sends a leakage function $L : \{0,1\}^K \rightarrow \{0,1\}^{t_R}$. It receives $L(\{(x_0^i, x_1^i)\}_{i\in[K]})$.

Let $(X, Y)$ be a random OT correlation (i.e., $X = (x_0, x_1), Y = (r, x_r)$, where $(x_0, x_1, r)$ are sampled uniformly at random.) We denote a $t_R$-leaky version of $(X, Y)^K$ described above as $((X, Y)^K)^{[t_R]}$.

**Definition 1 ($(n, p, t_R, \epsilon)$ One-Sided Malicious OT-Extractor).** *An $(n, p, t_R, \epsilon)$ one-sided malicious OT-extractor is an interactive protocol between 2 parties $S$ and $R$ with access to $((X, Y)^n)^{[t_R]}$ described above. The protocol implements $p$ independent copies of secure oblivious transfer instances with error $\epsilon$.*

In other words, we want the output oblivious transfer instances to satisfy the standard $\epsilon$-correctness and $\epsilon$-privacy requirements for OT. In more detail, the correctness requirement is that the receiver output is correct in all $p$ instances of OT with probability at least $(1 - \epsilon)$. The privacy requirement is that in every instance of the output OT protocol, a corrupt sender cannot output the receiver's choice bit, and a corrupt receiver cannot output the 'other message' of the sender with probability more than $\frac{1}{2} + \epsilon$.

**Theorem 3 (Extracting a Single OT).** *There exists a $(2\ell + 2n + 1, 1, \ell, 2^{-n})$ one-sided OT extractor according to Definition 1.*

**Theorem 4 (High Production Rate).** *There exists a $(2\ell + 2n, \frac{n}{\log^2 n}, \ell, \frac{1}{n \log n})$ one-sided OT extractor according to Definition 1.*

We prove these theorems by giving a construction and proof of security of such extractors in the following sections. We will make use of strong seeded extractors in our construction, and we define such extractors below.

**Definition 2 (Strong seeded extractors).** *A function* $\mathsf{Ext} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ *is called a strong seeded extractor for entropy* $k$ *if for any* $(n,k)$*-source* $X$ *and an independent random variable* $Y$ *that is uniform over* $\{0,1\}^d$*, it holds that* $(\mathsf{Ext}(X,Y),Y) \approx (U_m,Y)$.

Here, $U_m$ is a random variable that is uniformly distributed over $m$ bit strings and is independent of $Y$, namely $(U_m, Y)$ is a product distribution. In particular, it is known [12,18,34] how to construct strong seeded extractors for any entropy $k = \Omega(1)$ with seed length $d = O(\log n)$ and $m = 0.99\,k$ output bits.

*Construction.* In Fig. 4, we give the basic construction of an OT extractor that securely obtains a single oblivious transfer from $K = (2\ell + 2n)$ OTs, when a receiver can obtain at most $\ell$ bits of universal leakage from the joint distribution of sender inputs over all the OTs.

Let $\mathcal{E} : \{0,1\}^K \times \{0,1\}^n \to \{0,1\}$ be a strong randomness $(K, 2^{-n})$-extractor for seed length $d = O(n)$.
**Inputs:** Sender $\mathcal{S}$ has private inputs $(x_0, x_1) \in \{0,1\}^{2n}$ and receiver $\mathcal{R}$ has private input $\mathsf{b} \in \{0,1\}$.
**Given:** $K = 2\ell + 2n$ OTs, such that a malicious receiver can obtain additional $\ell$ bits of leakage on the joint distribution of all sender inputs.

1. **Invoking OT Correlations:**
   - For $i \in [K]$, $\mathcal{S}$ picks inputs $m_0^i, m_1^i \xleftarrow{\$} \{0,1\}$.
   - For $i \in [K]$, $\mathcal{S}$ invokes the $i^{th}$ OT on input $m_0^i, m_1^i$.
   - For $i \in [K]$, $\mathcal{R}$ invokes the $i^{th}$ OT on input (the same) choice bit $\mathsf{b}$.
2. **Sender Message:**
   - $\mathcal{S}$ picks random seed $s \xleftarrow{\$} \{0,1\}^d$ for the strong seeded extractor $\mathcal{E}$, and computes $M_0 = \mathcal{E}.\mathsf{Ext}(m_0^1 || m_0^2 || m_0^3 \ldots m_0^K, s)$ and $M_1 = \mathcal{E}.\mathsf{Ext}(m_1^1 || m_1^2 || m_1^3 \ldots m_1^K, s)$, where $||$ denotes the concatenation operator.
   - $\mathcal{S}$ sends $y_0 = M_0 \oplus x_0, y_1 = M_1 \oplus x_1$ to $\mathcal{R}$, along with seed $s$.
3. **Output:** $\mathcal{R}$ computes $x_\mathsf{b} = y_\mathsf{b} \oplus \mathcal{E}.\mathsf{Ext}(m_\mathsf{b}^1 || m_\mathsf{b}^2 || m_\mathsf{b}^3 \ldots || m_\mathsf{b}^K, s)$.

**Fig. 4.** $(2\ell + 2n, 1, \ell, 2^{-n})$ one-sided malicious correlation extractor.

Correctness is immediate from inspection. Intuitively, the protocol is secure against $\ell$ bits of universal (joint) leakage because setting $K = 2\ell + 2n$ still leaves $n$ bits of high entropy even when the receiver can obtain $2\ell + n$ bits of leakage. Moreover, with $\ell$ bits of additional universal leakage over all pairs of sender inputs $(m_0^1, m_1^1, m_0^2,\ m_1^2, \ldots m_0^K, m_1^K)$, the strong seeded extractor extracts an output that is statistically close to uniform, and this suffices to mask the sender input.

The formal proof of security can be found in the full version of this paper.

**High Production Rate:** It is possible to obtain an improved production rate at the cost of higher simulation error. This follows using techniques developed in prior work [17,36], and the details can be found in the full version.

## 6 UC Secure Computation in the Malicious Encapsulation Model

Let us consider the stateless protocol described in Sect. 3. In this protocol, the receiver must query $\mathsf{PUF}_s$ that he obtained from the sender on a random challenge $c$, before returning $\mathsf{PUF}_s$ to the sender. A malicious receiver cannot have queried $\mathsf{PUF}_s$ on both $c$ and $(c \oplus x_0 \oplus x_1)$, because $(x_0 \oplus x_1)$ is chosen by the sender, independently and uniformly at random, and is revealed *only after* the receiver has returned $\mathsf{PUF}_s$. If a malicious receiver was restricted to honestly returning the PUF generated by the sender, by unpredictability of $\mathsf{PUF}_s$, the output of $\mathsf{PUF}_s$ on $(c \oplus x_0 \oplus x_1)$ would be a completely unpredictable uniform random variable from the point of view of the receiver, and this sufficed to prove sender security.

However, if a malicious receiver had no such restriction, it could possibly generate a malicious PUF $\widehat{\mathsf{PUF}}$ of his own and give it to the sender, in place of the sender's PUF that it was actually supposed to return. The output of $\widehat{\mathsf{PUF}}$ would no longer remain unpredictable to the receiver and this would lead to a total break of security. As already pointed out in [7], this can be fixed by having the sender make "test queries" to the PUF he generates, before sending the PUF to the receiver. Indeed, when $\widehat{\mathsf{PUF}}$ is generated by the receiver independently of $\mathsf{PUF}_s$, the response of $\widehat{\mathsf{PUF}}$ on the sender's random test query will not match the response of $\mathsf{PUF}_s$ and the sender will catch such a cheating receiver with overwhelming probability.

However there could be a different attack: a malicious receiver can construct $\widehat{\mathsf{PUF}}$ encapsulating $\mathsf{PUF}_s$, such that $\widehat{\mathsf{PUF}}$ redirects all test queries to $\mathsf{PUF}_s$ (and outputs the value output by $\mathsf{PUF}_s$ on the evaluation query), whereas it maliciously answers all protocol queries. In order to rule this out, we ensure that the protocol queries (i.e., the input $c$ that the receiver must query $\mathsf{PUF}_s$ with) are generated uniformly at random, by using coin-tossing, combined with cut-and-choose tests to ensure that they are properly used. This is done carefully to ensure that the test queries and protocol queries are identically distributed in the view of $\widehat{\mathsf{PUF}}$ (and are revealed only after the receiver has sent $\widehat{\mathsf{PUF}}$ to the sender).

This ensures that if a maliciously generated $\widehat{\mathsf{PUF}}$ correctly answers all test queries, then with overwhelming probability it must necessarily have answered at least one evaluation query correctly according to the output of $\mathsf{PUF}_s$. At this point, an OT combiner is used to obtain one secure instance of OT.

Let the security parameter be $n$. The protocol in Fig. 5 UC-securely realizes 2-choose-1 OT in a stronger model, where a malicious party is allowed to create malicious PUFs that encapsulate other honest PUFs (see Sect. 2.1).

**Inputs:** Sender $\mathcal{S}$ has private inputs $(\mathsf{m}_0, \mathsf{m}_1) \in \{0,1\}^{2n}$ and Receiver $\mathcal{R}$ has private input $\mathsf{b} \in \{0,1\}$.

1. **Coin Flip I:** For $i \in [n]$, $\mathcal{R}$ picks $\mathsf{c}_1^i \xleftarrow{\$} \{0,1\}^n$, sends $\mathsf{d}^i = \mathsf{UC\text{-}Com.Commit}(\mathsf{c}_1^i)$ to $\mathcal{S}$. $\mathcal{S}$ chooses $\mathsf{c}_2^i \xleftarrow{\$} \{0,1\}^n$, sends $\mathsf{c}_2^i$ to $\mathcal{R}$. $\mathcal{R}$ computes $\mathsf{c}^i = \mathsf{c}_1^i \oplus \mathsf{c}_2^i$.
2. **Sender Message:** $\mathcal{S}$ generates $\mathsf{PUF}_s : \{0,1\}^n \to \{0,1\}^n$, and does:
   - **Test Queries:** For each $i \in [n]$, choose $\mathsf{TQ}_i \xleftarrow{\$} \{0,1\}^n$ and compute $\mathsf{TR}_i = \mathsf{PUF}_s(\mathsf{TQ}_i)$. Store the pair $(\mathsf{TQ}_i, \mathsf{TR}_i)$.
   - For each $i \in [n]$, choose a pair of random strings $(\mathsf{x}_0^i, \mathsf{x}_1^i) \xleftarrow{\$} \{0,1\}^{2n}$. Compute $(\mathsf{t}_0^i, \mathsf{t}_1^i) = \mathsf{UC\text{-}Com.Commit}(\mathsf{x}_0^i, \mathsf{x}_1^i)$. Send $(\mathsf{t}_0^i, \mathsf{t}_1^i)$ and $\mathsf{PUF}_s$ to $\mathcal{R}$.
3. **Receiver Message:** For each $i \in [n]$, choose a random string $(\mathsf{c}_0^i) \xleftarrow{\$} \{0,1\}^n$ and obtain $\mathsf{r}^i = \mathsf{PUF}_s(\mathsf{c}^i), \mathsf{r}_0^i = \mathsf{PUF}_s(\mathsf{c}_0^i)$. Abort if $\mathsf{PUF}_s$ aborts, else send $\mathsf{PUF}_s$ to $\mathcal{S}$. For $i \in [n]$, pick and send $(\hat{\mathsf{x}}_0^i, \hat{\mathsf{x}}_1^i) \xleftarrow{\$} \{0,1\}^{2n}$.
4. **Sender Message:** $\mathcal{S}$ does the following.
   - **Verification of TQ:** For each $i \in [n]$, if $\mathsf{TR}_i \neq \mathsf{PUF}_s(\mathsf{TQ}_i)$, abort.
   - For each $i \in [n]$, send $(\mathsf{x}_0^i, \mathsf{x}_1^i) = \mathsf{UC\text{-}Com.Decommit}(\mathsf{t}_0^i, \mathsf{t}_1^i)$ to $\mathcal{R}$.
5. **Receiver Message:** Abort if $\mathsf{UC\text{-}Com.Decommit}(\mathsf{t}_0^i, \mathsf{t}_1^i)$ does not verify for any $i \in [n]$. Else pick $\mathsf{b}_i \xleftarrow{\$} \{0,1\}$, compute and send $\mathsf{val}^i = \mathsf{c}^i \oplus \mathsf{x}_{\mathsf{b}_i}^i \oplus \hat{\mathsf{x}}_{\mathsf{b}_i}^i$ to $\mathcal{S}$.
6. **Cut-and-choose:**
   - **Coin Flip II:** $\mathcal{S}$ picks $r_{\mathcal{S}} \xleftarrow{\$} \{0,1\}^{2K}$, sends $t_{\mathcal{S}} = \mathsf{UC\text{-}Com.Commit}(r_{\mathcal{S}})$. $\mathcal{R}$ picks and sends $r_{\mathcal{R}} \xleftarrow{\$} \{0,1\}^{2K}$. $\mathcal{S}$ sends $r_{\mathcal{S}} = \mathsf{UC\text{-}Com.Decommit}(t_{\mathcal{S}})$, and $(\mathcal{S}, \mathcal{R})$ use $(r_{\mathcal{S}} \oplus r_{\mathcal{R}})$ to pick a subset $I$ of indices $i \in [n]$, of size $\frac{K}{2}$.
   - For $i \in [I]$, $\mathcal{R}$ sends $\mathsf{c}_1^i = \mathsf{UC\text{-}Com.Decommit}(\mathsf{d}^i)$.
   - **Verification:** $\mathcal{S}$ computes $\mathsf{c}^i = \mathsf{c}_1^i \oplus \mathsf{c}_2^i$ and checks if either $\mathsf{val}^i = \mathsf{c}^i \oplus \mathsf{x}_0^i \oplus \hat{\mathsf{x}}_0^i$ OR $\mathsf{val}^i = \mathsf{c}^i \oplus \mathsf{x}_1^i \oplus \hat{\mathsf{x}}_1^i$. If not, $\mathcal{S}$ aborts.
7. **Receiver Message:** For each $i \in [n] \setminus I$, $\mathcal{R}$ sends $\mathsf{bc}_i = \mathsf{b}_i \oplus \mathsf{b}$ to $\mathcal{S}$.
8. **Sender Message:** $\mathcal{S}$ computes $\mathsf{S}_0 = \mathsf{m}_0 \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(\mathsf{val}^i \oplus \mathsf{x}_{\mathsf{bc}_i}^i \oplus \hat{\mathsf{x}}_{\mathsf{bc}_i}^i)$, $\mathsf{S}_1 = \mathsf{m}_1 \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(\mathsf{val}^i \oplus \mathsf{x}_{1-\mathsf{bc}_i}^i \oplus \hat{\mathsf{x}}_{1-\mathsf{bc}_i}^i)$. $\mathcal{S}$ sends $(\mathsf{S}_0, \mathsf{S}_1)$ to $\mathcal{R}$.

**Outputs:** $\mathcal{S}$ has no output. $\mathcal{R}$ outputs $\mathsf{m}_{\mathsf{b}} := (\mathsf{S}_{\mathsf{b}} \oplus \mathsf{r}^1 \oplus \ldots \oplus \mathsf{r}^n)$.

**Fig. 5.** OT in the malicious stateless PUF model with encapsulation. We underline all differences from the protocol in the stateless malicious PUF model.

We emphasize that our protocol does not require that honest parties must have the capability to encapsulate PUFs, yet it is secure even when adversarial parties can create encapsulated PUFs. The protocol uses a UC-commitment scheme, secure in the malicious stateless encapsulated PUF model. We use $\mathsf{Com}$ to denote the ideal functionality for such a scheme. We construct such a scheme in Sect. 7.

Though the commitment scheme we construct is UC-secure, it is not immediately clear that it composes with the rest of the OT protocol for the same reasons as were described in Sect. 4. Namely, the UC framework seemingly does not capture the possibility of transfer of PUFs across sub-protocols, thus we would like to ensure that our UC-commitment scheme composes with the rest of the protocol even if PUFs created for the commitment scheme are used elsewhere.

Like in Sect. 4, this can be resolved by requiring both parties to return PUFs back to the respective creators at the end of the decommitment phase, and the creators performing simple verification checks to ensure that the correct PUF was returned. If any party fails to return the PUF, the other party aborts the protocol. Therefore, parties cannot pass off PUFs used by some party in a previous sub-protocol as a new PUF in a different sub-protocol.

*Correctness*

*Claim.* For all $(m_0, m_1) \in \{0,1\}^2$ and $\mathsf{b} \in \{0,1\}$, the output of $\mathcal{R}$ equals $m_b$.

*Proof.* If $b = 0$, $\mathsf{bc}^i = \mathsf{b}^i$ for all $i$, and the receiver computes:

$$\begin{aligned} m_0' &= S_0 \bigoplus_{i \in n \setminus I} r^i = S_0 \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(c^i) = S_0 \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(\mathsf{val}^i \oplus x^i_{\mathsf{b}_i}) \\ &= m_0 \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(\mathsf{val}^i \oplus x^i_{\mathsf{bc}_i}) \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(\mathsf{val}^i \oplus x^i_{\mathsf{b}_i}) = m_0. \end{aligned}$$

If $b = 1$, $1 - \mathsf{bc}^i = \mathsf{b}^i$ for all $i$, and the receiver computes:

$$\begin{aligned} m_1' &= S_1 \bigoplus_{i \in n \setminus I} r^i = S_1 \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(c^i) = S_1 \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(\mathsf{val}^i \oplus x^i_{b_i}) \\ &= m_1 \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(\mathsf{val}^i \oplus x^i_{1-\mathsf{bc}_i}) \bigoplus_{i \in n \setminus I} \mathsf{PUF}_s(\mathsf{val}^i \oplus x^i_{\mathsf{b}_i}) = m_1. \end{aligned}$$

The formal proof of security can be found in the full version of the paper.

## 7 UC Commitments in the Malicious Encapsulation Model

In this section we construct unconditional UC commitments using stateless PUFs. The model we consider is incomparable with respect to the one of [8] since in our model an adversary can encapsulate honest PUFs (see Sect. 2.1) when creating malicious *stateless encapsulated* PUFs. Note that the protocol does not require any honest party to have the ability to encapsulate PUFs, but is secure against parties that do have this ability.

We note that it suffices to construct an extractable commitment scheme that is secure against encapsulation. Indeed, given such a scheme, Damgård and Scafuro [8] show that it is possible to compile the extractable commitment scheme

using an additional ideal commitment scheme, to obtain a UC commitment scheme that is secure in the malicious stateless PUF model. Since the compiler of [8] does not require any additional PUFs at all, if the extractable commitment and the ideal commitment are secure against encapsulation attacks, then so is the resulting UC commitment.

*Extractable Commitments.* We describe how to construct an extractable bit commitment scheme $\mathsf{ExtCom} = (\mathsf{ExtCom.Commit}, \mathsf{ExtCom.Decommit}, \mathsf{ExtCom.Extract})$ that is secure in the malicious stateless PUFs model with encapsulation. We start with the extractable commitment scheme of [8] that is secure against malicious PUFs in the non-encapsulated setting. They crucially rely on the fact that the initial PUF (let's call it $\mathsf{PUF}_r$) sent by the receiver can not be replaced by the committer (as that would be caught using a previously computed test query). To perform extraction, the simulator against a malicious committer observes the queries made by the committer to $\mathsf{PUF}_r$ and extracts the committer's bit. However, in the encapsulated setting, the malicious committer could encapsulate the receiver's PUF inside another PUF (let's call it $\widehat{\mathsf{PUF}_r}$) that, for all but one query, answers with the output of $\mathsf{PUF}_r$. For the value that the committer is actually required to query on, $\widehat{\mathsf{PUF}_r}$ responds with a maliciously chosen value. Observe that in the protocol description, this query is chosen only by the committer and hence this is an actual attack. Therefore, except with negligible probability, all the receiver's test queries will be answered by $\widehat{\mathsf{PUF}_r}$ with the output of the receiver's original PUF $\mathsf{PUF}_r$. On the other hand, since the target query is no longer forwarded by $\widehat{\mathsf{PUF}_r}$ to the receiver's original PUF, the simulator does not get access to the target query and hence can not extract the committer's bit.

To overcome this issue, we develop a new technique that forces the malicious committer to reveal the target query to the simulator (but not to the honest receiver). After the committer returns $\widehat{\mathsf{PUF}_r}$, the receiver creates a new PUF (let's call it $\mathsf{PUF}_R$). Now, using the commitment, the receiver queries $\mathsf{PUF}_R$ on two values, one of which is guaranteed to be the output of $\widehat{\mathsf{PUF}_r}$ on the target query. The receiver stores these two outputs and sends $\mathsf{PUF}_R$ to the committer. The malicious committer now has to query $\mathsf{PUF}_R$ with $\widehat{\mathsf{PUF}_r}$'s output on his target query and commit to the value that is given in output by $\mathsf{PUF}_R$ (using an ideal commitment scheme). In the decommitment phase, using the previously stored values and the committer's input bit, the receiver can verify that the committer indeed queried $\mathsf{PUF}_R$ on the correct value. Observe that since the receiver has precomputed the desired output, the malicious committer will not be able to produce an honest decommitment if he tampers with $\mathsf{PUF}_R$ and produces a different output. Therefore, the malicious committer *must* indeed query $\mathsf{PUF}_R$ and this can be observed by the simulator and used to extract the committer's bit. Our scheme is described in Fig. 6. We show that this scheme is correct, statistically hiding, and extractable; and give further details in the full version.

**Inputs:** Committer $\mathcal{C}$ has private input $\mathsf{b} \in \{0,1\}$ and receiver $\mathcal{R}$ has no input.
**Commitment Phase:**

1. **Receiver Message:** $\mathcal{R}$ does the following:
   - Generate a PUF $\mathsf{PUF}_r : \{0,1\}^{3n} \to \{0,1\}^{3n}$.
   - **Test Queries :** For each $i \in [n]$, choose $\mathsf{TQ}_i \xleftarrow{\$} \{0,1\}^{3n}$, and compute $\mathsf{TR}_i = \mathsf{PUF}_r(\mathsf{TQ}_i)$. Store the pair $(\mathsf{TQ}_i, \mathsf{TR}_i)$. Send $\mathsf{PUF}_r$ to $\mathcal{C}$.
2. **Committer Message:** $\mathcal{C}$ does the following:
   - Generate a PUF $\mathsf{PUF}_s : \{0,1\}^n \to \{0,1\}^{3n}$.
   - For each $i \in [n]$, choose $\mathsf{s}_i \in \{0,1\}^n$. Compute $\sigma_{s_i} = \mathsf{PUF}_s(\mathsf{s}_i)$ and $\sigma_{r_i} = \mathsf{PUF}_r(\sigma_{s_i})$. Send $\mathsf{PUF}_s, \mathsf{PUF}_r$ to $\mathcal{R}$.
3. **Receiver Message:** $\mathcal{R}$ does the following:
   - **Verification :** For each $i \in [n]$, if $\mathsf{TR}_i \neq \mathsf{PUF}_r(\mathsf{TQ}_i)$, abort.
   - For each $i \in [n]$, choose a random string $\mathsf{r}_i \xleftarrow{\$} \{0,1\}^{3n}$ and send $\mathsf{r}_i$ to $\mathcal{C}$.
4. **Committer Message:** $\mathcal{C}$ does the following: If $\mathsf{b} = 0$, set $\mathsf{c}_i = \sigma_{s_i}$ for $i \in [n]$, else set $\mathsf{c}_i = (\sigma_{s_i} \oplus \mathsf{r}_i)$ for $i \in [n]$. Send $\mathsf{c}_i$ to $\mathcal{R}$.
5. **Receiver Message:** $\mathcal{R}$ does the following:
   - Generate a PUF $\mathsf{PUF}_R : \{0,1\}^{3n} \to \{0,1\}^{3n}$.
   - For $i \in [n]$, set $\mathsf{y}_i = \mathsf{PUF}_R(\mathsf{PUF}_r(\mathsf{c}_i)), \mathsf{z}_i = \mathsf{PUF}_R(\mathsf{PUF}_r(\mathsf{c}_i \oplus \mathsf{r}_i))$, send $\mathsf{PUF}_R$ to $\mathcal{C}$.
6. **Committer Message:** For each $i \in [n]$, $\mathcal{C}$ computes $\mathsf{x}_i = \mathsf{PUF}_R(\sigma_{r_i})$. $\mathcal{C}$ computes and sends $\mathsf{t}_i = \mathsf{IdealCom.Commit}(\mathsf{x}_i)$ to $\mathcal{R}$.

**Decommitment Phase:**

1. **Committer Message:** $\mathcal{C}$ does the following:
   - Send $\mathsf{b}$ to $\mathcal{R}$ and for each $i \in [n]$, send $\mathsf{s}_i$, $\mathsf{IdealCom.Decommit}(\mathsf{x}_i)$ to $\mathcal{R}$.
2. **Receiver** $\mathcal{R}$ does the following:
   - For any $i \in [n]$, if $\mathsf{IdealCom.Decommit}(\mathsf{x}_i)$ does not verify, output $\perp$.
   - If $\mathsf{b} = 0$, $\mathsf{c}_i = \mathsf{PUF}_s(\mathsf{s}_i)$ and $\mathsf{x}_i = \mathsf{y}_i$ for all $i \in [n]$, output 0, else $\perp$.
   - If $\mathsf{b} = 1$, $\mathsf{c}_i = (\mathsf{PUF}_s(\mathsf{s}_i) \oplus \mathsf{r}_i)$ and $\mathsf{x}_i = \mathsf{z}_i$ for all $i \in [n]$, output 1, else $\perp$.

**Fig. 6.** Protocol for Extractable Commitment in the malicious stateless PUF model with encapsulation.

**Acknowledgements.** We thank the anonymous reviewers for valuable comments, and in particular for suggesting some important updates to our functionality for encapsulated PUFs.

## A Formal Models for PUFs

While we discuss the physical behaviour of PUFs, and their various properties in detail in the full version of the paper, here, we describe the formal modelling of various honest, malicious and encapsulating PUFs.

We model honest PUFs similar to prior work. The ideal functionality for honest PUFs is described in Fig. 7. We assume that in situations where $P_i$ is required to send a message of the form $(\ldots, P_i, \ldots)$, the ideal functionality checks

$\mathcal{F}_{\mathsf{HPUF}}$ uses PUF family $\mathcal{P} = (\mathsf{Sample}, \mathsf{Eval})$ with parameters $(rg, d_{\mathsf{noise}}, d_{\mathsf{min}}, m)$. It runs on input the security parameter $1^K$, with parties $\mathbb{P} = \{P_1, \ldots, P_n\}$ and adversary $\mathcal{S}$.

- When a party $\hat{P} \in \mathbb{P} \cup \{\mathcal{S}\}$ writes $(\mathsf{init}_{\mathsf{PUF}}, \mathsf{sid}, \hat{P})$ on the input tape of $\mathcal{F}_{\mathsf{HPUF}}$, $\mathcal{F}_{\mathsf{HPUF}}$ checks whether $\mathcal{L}$ already contains a tuple $(\mathsf{sid}, *, *, *, *)$:
  - If this is the case, then turn into the waiting state.
  - Else, draw $\mathsf{id} \leftarrow \mathsf{Sample}_{\mathtt{mode}}(1^K)$ from the PUF family. Put $(\mathsf{sid}, \mathsf{id}, \hat{P}, \mathsf{notrans})$ in $\mathcal{L}$ and write $(\mathsf{initialized}_{\mathsf{PUF}}, \mathsf{sid})$ on the input tape of $\hat{P}$.
- When party $P_i$ writes $(\mathsf{eval}_{\mathsf{PUF}}, \mathsf{sid}, P_i, q)$ on $\mathcal{F}_{\mathsf{HPUF}}$'s input tape, $\mathcal{F}_{\mathsf{HPUF}}$ checks if there exists a tuple $(\mathsf{sid}, \mathsf{id}, P_i, \mathsf{notrans})$ in $\mathcal{L}$.
  - If not, then turn into waiting state.
  - Else, run $a \leftarrow \mathsf{Eval}_{\mathtt{mode}}(1^K, \mathsf{id}, q)$. Write $(\mathsf{response}_{\mathsf{PUF}}, \mathsf{sid}, q, a)$ on $P_i$'s input tape.
- When a party $P_i$ sends $(\mathsf{handover}_{\mathsf{PUF}}, \mathsf{sid}, P_i, P_j)$ to $\mathcal{F}_{\mathsf{HPUF}}$, check if there exists a tuple $(\mathsf{sid}, *, P_i, \mathsf{notrans})$ in $\mathcal{L}$.
  - If not, then turn into waiting state.
  - Else, modify the tuple $(\mathsf{sid}, \mathsf{id}, P_i, \mathsf{notrans})$ to the updated tuple $(\mathsf{sid}, \mathsf{id}, \perp, \mathsf{trans}(P_j))$. Write $(\mathsf{invoke}_{\mathsf{PUF}}, \mathsf{sid}, P_i, P_j)$ on $P_i$'s input tape.
- When the adversary sends $(\mathsf{eval}_{\mathsf{PUF}}, \mathsf{sid}, P_i, q)$ to $\mathcal{F}_{\mathsf{HPUF}}$, check if $\mathcal{L}$ contains a tuple $(\mathsf{sid}, \mathsf{id}, \perp, \mathsf{trans}(*))$.
  - If not, then turn into waiting state.
  - Else, run $a \leftarrow \mathsf{Eval}_{\mathtt{mode}}(1^K, \mathsf{id}, q)$ and return $(\mathsf{response}_{\mathsf{PUF}}, \mathsf{sid}, q, a)$ to $P_i$.
- When the adversary sends $(\mathsf{ready}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ to $\mathcal{F}_{\mathsf{HPUF}}$, check if $\mathcal{L}$ contains the tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(P_j))$.
  - If not found, turn into the waiting state.
  - Else, change the tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(P_j))$ to $(\mathsf{sid}, \mathsf{id}, P_i, \mathsf{notrans})$ and write $(\mathsf{handover}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ on $P_j$'s input tape and store the tuple $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$.
- When the adversary sends $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ to $\mathcal{F}_{\mathsf{HPUF}}$, check if the tuple $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ has been stored. If not, return to the waiting state. Else, write this tuple to the input tape of $P_i$.

**Fig. 7.** The ideal functionality $\mathcal{F}_{\mathsf{HPUF}}$ for honest PUFs.

that the message is indeed coming from party $P_i$, if not the ideal functionality $\mathcal{F}_{\mathsf{HPUF}}$ turns into waiting state.

*Modeling Malicious PUFs.* We model malicious PUFs as in [27]. Their ideal functionality is parameterized by two PUF families in order to handle honestly and maliciously generated PUFs: The honestly generated family is a pair $(\mathsf{Sample}_{\mathsf{normal}}, \mathsf{Eval}_{\mathsf{normal}})$ and the malicious one is $(\mathsf{Sample}_{\mathsf{mal}}, \mathsf{Eval}_{\mathsf{mal}})$. Whenever a party $P_i$ initializes a PUF, then it specifies if it is an honest or a malicious PUF by sending $\mathsf{mode} \in \{\mathsf{nor}, \mathsf{mal}\}$ to the functionality $\mathcal{F}_{\mathsf{PUF}}$. The ideal functionality then initialises the appropriate PUF family and it also stores a tag nor or mal representing this family. Whenever the PUF is evaluated, the ideal functionality uses the evaluation algorithm that corresponds to the tag.

The handover procedure is identical to the original formulation of Brzuska et al., where each PUF has a status $\mathsf{flag} \in \{\mathsf{trans}(\mathcal{R}), \mathsf{notrans}\}$ that indicates if a PUF is in transit or not. A PUF that is in transit can be queried by the adversary. Thus, whenever a party $P_i$ sends a PUF to $P_j$, then the status flag is changed from notrans to trans and the attacker can evaluate the PUF. At some point, the attacker sends $\mathsf{ready}_{\mathsf{PUF}}$ to the ideal functionality to indicate that it is not querying the PUF anymore. The ideal functionality then hands the PUF over to $P_j$ and changes the status flag back to notrans. The party $P_j$ may evaluate the PUF. Finally, when the attacker sends the message $\mathsf{received}_{\mathsf{PUF}}$ to the ideal functionality, then $\mathcal{F}_{\mathsf{PUF}}$ sends $\mathsf{received}_{\mathsf{PUF}}$ to $P_i$ in order to notify $P_i$ that the handover is over. The ideal functionality for malicious PUFs is shown in Fig. 8. We refer the reader to [27] for more details on the different properties of malicious PUFs.

We additionally allow malicious PUFs to maintain $\mathsf{poly}(n)$ a-prior bounded memory. This is done by allowing $\mathsf{Eval}_{\mathsf{mal}}$ to be a stateful procedure.

$\mathcal{F}_{\mathsf{MPUF}}$ uses PUF families $\mathcal{P}_1 = (\mathsf{Sample}_{\mathtt{normal}}, \mathsf{Eval}_{\mathtt{normal}})$ with parameters $(rg, d_{\mathsf{noise}}, d_{\mathsf{min}}, m)$, and $\mathcal{P}_2 = (\mathsf{Sample}_{\mathtt{mal}}, \mathsf{Eval}_{\mathtt{mal}})$. It runs on input the security parameter $1^K$, with parties $\mathbb{P} = \{P_1, \ldots, P_n\}$ and adversary $\mathcal{S}$.

- When a party $\hat{P} \in \mathbb{P} \cup \{\mathcal{S}\}$ writes $(\mathsf{init}_{\mathsf{PUF}}, \mathsf{sid}, \mathtt{mode}, \hat{P})$ on the input tape of $\mathcal{F}_{\mathsf{MPUF}}$, where $\mathtt{mode} \in \{\mathtt{normal}, \mathtt{mal}\}$, then $\mathcal{F}_{\mathsf{MPUF}}$ checks whether $\mathcal{L}$ already contains a tuple $(\mathsf{sid}, *, *, *, *)$: If this is the case, then turn into the waiting state. Else, draw $\mathsf{id} \leftarrow \mathsf{Sample}_{\mathtt{mode}}(1^K)$ from the PUF family. Put $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \hat{P}, \mathsf{notrans})$ in $\mathcal{L}$ and write $(\mathsf{initialized}_{\mathsf{PUF}}, \mathsf{sid})$ on the input tape of $\hat{P}$.
- When party $P_i \in \mathbb{P}$ writes $(\mathsf{eval}_{\mathsf{PUF}}, \mathsf{sid}, P_i, q)$ on $\mathcal{F}_{\mathsf{MPUF}}$'s input tape, check if there exists a tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$ in $\mathcal{L}$. If not, then turn into waiting state. Else, run $a \leftarrow \mathsf{Eval}_{\mathtt{mode}}(1^K, \mathsf{id}, q)$. Write $(\mathsf{response}_{\mathsf{PUF}}, \mathsf{sid}, q, a)$ on $P_i$'s input tape.
- When a party $P_i$ sends $(\mathsf{handover}_{\mathsf{PUF}}, \mathsf{sid}, P_i, P_j)$ to $\mathcal{F}_{\mathsf{PUF}}$, check if there exists a tuple $(\mathsf{sid}, *, *, P_i, \mathsf{notrans})$ in $\mathcal{L}$. If not, then turn into waiting state. Else, modify the tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$ to the updated tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(P_j))$. Write $(\mathsf{invoke}_{\mathsf{PUF}}, \mathsf{sid}, P_i, P_j)$ on $P_i$'s input tape to indicate that a handover occurred between $P_i$ and $P_j$.
- When the adversary sends $(\mathsf{eval}_{\mathsf{PUF}}, \mathsf{sid}, P_i, q)$ to $\mathcal{F}_{\mathsf{MPUF}}$, check if $\mathcal{L}$ contains a tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(*))$ or $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$. If not, then turn into waiting state. Else, run $a \leftarrow \mathsf{Eval}_{\mathtt{mode}}(1^K, \mathsf{id}, q)$ and return $(\mathsf{response}_{\mathsf{PUF}}, \mathsf{sid}, q, a)$ to $P_i$.
- When the adversary sends $(\mathsf{ready}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ to $\mathcal{F}_{\mathsf{MPUF}}$, check if $\mathcal{L}$ contains the tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(P_j))$. If not found, turn into the waiting state. Else, change the tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(P_j))$ to $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_j, \mathsf{notrans})$ and write $(\mathsf{handover}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ on $P_j$'s input tape and store the tuple $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$.
- When the adversary sends $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ to $\mathcal{F}_{\mathsf{MPUF}}$, check if the tuple $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ has been stored. If not, return to the waiting state. Else, write this tuple to the input tape of $P_i$.

**Fig. 8.** The ideal functionality $\mathcal{F}_{\mathsf{MPUF}}$ for malicious PUFs.

$\mathcal{F}_{\text{E-PUF}}$ uses PUF families $\mathcal{P}_1 = (\mathsf{Sample}_{\mathtt{normal}}, \mathsf{Eval}_{\mathtt{normal}})$ with parameters $(rg, d_{\text{noise}}, d_{\min}, m)$, and $\mathcal{P}_2 = (\mathsf{Sample}_{\mathtt{mal}}, \mathsf{Eval}_{\mathtt{mal}})$. It runs on input the security parameter $1^K$, with parties $\mathbb{P} = \{P_1, \ldots, P_n\}$ and adversary $\mathcal{S}$ corrupting some parties.

- When a party $P_i \in \mathbb{P} \cup \{\mathcal{S}\}$ writes $(\mathsf{init}_{\mathsf{PUF}}, \mathsf{sid}, \mathtt{mode}, P_i)$ on the input tape of $\mathcal{F}_{\text{E-PUF}}$, where $\mathtt{mode} \in \{\mathtt{normal}, \mathtt{mal}\}$, then $\mathcal{F}_{\text{E-PUF}}$ checks whether $\mathcal{L}$ already contains a tuple $(\mathsf{sid}, \mathsf{id}, *, *, *, *)$ for some $\mathsf{id}$. If it does, turn to waiting state. Else, draw $\mathsf{id} \leftarrow \mathsf{Sample}_{\mathtt{mode}}(1^K)$ from the PUF family. Put $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$ in $\mathcal{L}$ and write $(\mathsf{initialized}_{\mathsf{PUF}}, \mathsf{sid})$ on the input tape of $P_i$. If any of the checks failed, turn to waiting state.
- When the adversary $P_i$ writes $\mathsf{reassign}(\mathsf{sid}, \mathsf{sid}', P_i)$ on the input tape of $\mathcal{F}_{\text{E-PUF}}$, check if there exists a tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$, and check that $\mathcal{L}$ does not already contains a tuple $(\mathsf{sid}, \mathsf{id}, *, *, *, *)$ for some $\mathsf{id}$. If either of the conditions are not met, turn to waiting state. Else, replace the first tuple with $(\mathsf{sid}', \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$.
- When the adversary $P_i$ writes $(\mathsf{encap}_{\mathsf{PUF}}, \mathsf{sid}, \mathsf{sid}', P_i)$ on the input tape of $\mathcal{F}_{\text{E-PUF}}$, check if there exist tuples $(\mathsf{sid}, *, *, P_i, \mathsf{notrans})$ and $(\mathsf{sid}', *, *, P_i, \mathsf{notrans})$. If such tuples exist, set $\mathsf{owner}(\mathsf{sid}) = \mathsf{sid}'$ [a].
- When party $P_i$ sends $(\mathsf{handover}_{\mathsf{PUF}}, \mathsf{sid}, P_i, P_j)$ to $\mathcal{F}_{\text{E-PUF}}$, check if there exists a tuple $(\mathsf{sid}, *, *, P_i, \mathsf{notrans})$ in $\mathcal{L}$. If not, then turn into waiting state. Else, modify the tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$ to $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(P_j))$. Write $(\mathsf{invoke}_{\mathsf{PUF}}, \mathsf{sid}, P_i, P_j)$ on $P_i$'s input tape [b].
- When a party $P_i \in \mathbb{P} \cup \{\mathcal{S}\}$ writes $(\mathsf{eval}_{\mathsf{PUF}}, \mathsf{sid}, P_i, q)$ on $\mathcal{F}_{\text{E-PUF}}$'s input tape, check if there exists a tuple $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$ or $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(*))$ in $\mathcal{L}$. If not, then turn into waiting state. Else, run $a \leftarrow \mathsf{Eval}_{\mathtt{mode}}(1^K, \mathsf{id}, q)$. Write $(\mathsf{response}_{\mathsf{PUF}}, \mathsf{sid}, q, a)$ on $P_i$'s input tape.
- The $\mathsf{Eval}_{\mathtt{mal}}$ procedure can either makes calls to $\mathsf{Eval}_{\mathtt{normal}}$, or can write $(\mathsf{eval}_{\mathsf{PUF}}, \mathsf{sid}*, \mathsf{sid}, q*)$ on $\mathcal{F}_{\text{E-PUF}}$'s input tape. If $\mathsf{Eval}_{\mathtt{mal}}$ writes $(\mathsf{eval}_{\mathsf{PUF}}, \mathsf{sid}*, \mathsf{sid}, q*)$ on $\mathcal{F}_{\mathcal{F}_{\text{E-PUF}}}$'s input tape, check if $\mathsf{owner}(\mathsf{sid}*) = \mathsf{sid}$. If not, turn to waiting state. Else, like the previous bullet, check if there exists a tuple $(\mathsf{sid}*, \mathsf{id}, \mathtt{mode}, P_i, \mathsf{notrans})$ or $(\mathsf{sid}*, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(*))$ in $\mathcal{L}$. If not, then turn into waiting state. Else, run $a \leftarrow \mathsf{Eval}_{\mathtt{mode}}(1^K, \mathsf{id}, q)$ and return $(\mathsf{response}_{\mathsf{PUF}}, \mathsf{sid}*, q, a)$ as output to $\mathsf{sid}$.
- When the adversary sends $(\mathsf{ready}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ to $\mathcal{F}_{\text{E-PUF}}$, check if $\mathcal{L}$ contains $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(P_j))$. If not, turn into waiting state. Else, change $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, \perp, \mathsf{trans}(P_j))$ to $(\mathsf{sid}, \mathsf{id}, \mathtt{mode}, P_j, \mathsf{notrans})$, write $(\mathsf{handover}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ on $P_j$'s input tape and store $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$.
- When the adversary sends $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ to $\mathcal{F}_{\text{E-PUF}}$, check if $(\mathsf{received}_{\mathsf{PUF}}, \mathsf{sid}, P_i)$ has been stored. If not, return to waiting state. Else, write this tuple to the input tape of $P_i$.

[a] Intuitively, when a (malicious) party encapsulates a PUF, this sets the outer PUF as owner of the inner PUF. Even the adversary can access the inner PUF via evaluation queries to outer PUF. This step permits multiple iterative encapsulations.

[b] Handover does not change the owner (outer PUF) of an (inner) encapsulated PUF.

**Fig. 9.** The ideal functionality $\mathcal{F}_{\text{E-PUF}}$ for malicious PUFs that may *encapsulate* PUFs.

*Modeling Encapsulating PUFs.* We model malicious PUFs that can encapsulate functionalities as in [6,27]. This functionality formalizes the intuition that an honest user can create a PUF implementing a random function, but an adversary given the PUF can only observe its input/output characteristics.

$\mathcal{F}_{\mathsf{E\text{-}PUF}}$ models the PUF (sent by party $P_i$ to party $P_j$) encapsulating some functionality $M_{ij}$. The changes from the previous definition [27] that we make is that $M_{ij}$ is now an oracle machine (instead of a functionality) which can make evaluation calls to other PUFs itself. The ideal functionality for malicious PUFs that could possibly encapsulate honest PUFs, is described in Fig. 9. $\mathcal{F}_{\mathsf{E\text{-}PUF}}$ models the following sequence of events: (1) a party $P_i$ samples a random PUF from the challenge space, (2) $P_i$ then gives this PUF to another party $P_j$ (the receiver) who can use the PUF as a black-box implementing $M_{ij}$, (3) On giving $M_{ij}$, $P_i$ loses oracle access to all PUFs of which it was previously the owner but which $M_{ij}$ has oracle access to. Figure 9 has the formal description of $\mathcal{F}_{\mathsf{E\text{-}PUF}}$ based on such an algorithm $M_{ij}$.

We assume that every PUF has a *single* calling procedure known as its *owner*. This owner can either be a party, or another PUF (in the case of adversarially generated PUFs). This models (refer to the first bullet in Fig. 9) the fact that an adversary that receives a PUF implementing $M_{xy}$ can either keep the PUF to make calls later or incorporate the functionality of this PUF in a black-box manner into another (maliciously created) PUF, but cannot do both. The evaluation procedure for a malicious encapsulating outer PUF, carefully checks that the outer PUF has ownership of inner PUFs (refer the second bullet in Fig. 9), before allowing the malicious outer evaluation procedure oracle access to any inner PUF. The handover operation (described in the third bullet in Fig. 9) is similarly carefully modified to ensure that the party that receives an encapsulated PUF can only access the inner PUF via evaluation queries to the outer PUF. Each PUF is uniquely identified by an identifier known as $\mathsf{id}$.

Finally, we note that our model may also allow an adversary to "dismount" a PUF, i.e., separate out its inner component PUFs. For simplicity, we choose to not formalize this requirement. Our protocols trivially remain secure in this model since we never require the honest parties to hand over any "encap"-PUFs back to the adversary, where an "encap"-PUF is a malicious PUF that may be encapsulating honest PUFs.

## References

1. Agrawal, S., Ananth, P., Goyal, V., Prabhakaran, M., Rosen, A.: Lower bounds in the hardware token model. In: Lindell, Y. (ed.) TCC 2014. LNCS, vol. 8349, pp. 663–687. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54242-8_28
2. Armknecht, F., Moriyama, D., Sadeghi, A.-R., Yung, M.: Towards a unified security model for physically unclonable functions. In: Sako, K. (ed.) CT-RSA 2016. LNCS, vol. 9610, pp. 271–287. Springer, Cham (2016). doi:10.1007/978-3-319-29485-8_16
3. Boureanu, I., Ohkubo, M., Vaudenay, S.: The limits of composable crypto with transferable setup devices. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS 2015, Singapore, 14–17 April 2015, pp. 381–392. ACM (2015)

4. Brzuska, C., Fischlin, M., Schröder, H., Katzenbeisser, S.: Physically uncloneable functions in the universal composition framework. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 51–70. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22792-9_4
5. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Foundations of Computer Science (FOCS 2001), pp. 136–145 (2001)
6. Chandran, N., Goyal, V., Sahai, A.: New constructions for UC secure computation using tamper-proof hardware. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 545–562. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78967-3_31
7. Dachman-Soled, D., Fleischhacker, N., Katz, J., Lysyanskaya, A., Schröder, D.: Feasibility and infeasibility of secure computation with malicious PUFs. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 405–420. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44381-1_23
8. Damgård, I., Scafuro, A.: Unconditionally secure and universally composable commitments from physical assumptions. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013. LNCS, vol. 8270, pp. 100–119. Springer, Heidelberg (2013). doi:10.1007/978-3-642-42045-0_6
9. Dodis, Y., Ostrovsky, R., Reyzin, L., Smith, A.: Fuzzy extractors: how to generate strong keys from biometrics and other noisy data. SIAM J. Comput. **38**(1), 97–139 (2008)
10. Döttling, N., Kraschewski, D., Müller-Quade, J., Nilges, T.: General statistically secure computation with bounded-resettable hardware tokens. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9014, pp. 319–344. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46494-6_14
11. Döttling, N., Mie, T., Müller-Quade, J., Nilges, T.: Implementing resettable UC-functionalities with untrusted tamper-proof hardware-tokens. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 642–661. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36594-2_36
12. Dvir, Z., Kopparty, S., Saraf, S., Sudan, M.: Extensions to the method of multiplicities, with applications to Kakeya sets and mergers. SIAM J. Comput. **42**(6), 2305–2328 (2013)
13. Eichhorn, I., Koeberl, P., van der Leest, V.: Logically reconfigurable PUFs: memory-based secure key storage. In: Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing, STC 2011, pp. 59–64. ACM, New York (2011)
14. Goyal, V., Ishai, Y., Sahai, A., Venkatesan, R., Wadia, A.: Founding cryptography on tamper-proof hardware tokens. In: Micciancio, D. (ed.) TCC 2010. LNCS, vol. 5978, pp. 308–326. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11799-2_19
15. Goyal, V., Maji, H.K.: Stateless cryptographic protocols. In: Ostrovsky, R. (ed.) IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, 22–25 October 2011, pp. 678–687. IEEE Computer Society (2011)
16. Guajardo, J., Kumar, S.S., Schrijen, G.-J., Tuyls, P.: FPGA intrinsic PUFs and their use for IP protection. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 63–80. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74735-2_5
17. Gupta, D., Ishai, Y., Maji, H.K., Sahai, A.: Secure computation from leaky correlated randomness. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 701–720. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48000-7_34
18. Guruswami, V., Umans, C., Vadhan, S.P.: Unbalanced expanders and randomness extractors from Parvaresh-Vardy codes. J. ACM **56**(4) (2009)

19. Hazay, C., Lindell, Y.: Constructions of truly practical secure protocols using standardsmartcards. In: Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, 27–31 October 2008, pp. 491–500 (2008)
20. Hazay, C., Polychroniadou, A., Venkitasubramaniam, M.: Composable security in the tamper-proof hardware model under minimal complexity. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9985, pp. 367–399. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53641-4_15
21. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Extracting correlations. In: 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, Atlanta, Georgia, USA, 25–27 October 2009, pp. 261–270. IEEE Computer Society (2009)
22. Järvinen, K., Kolesnikov, V., Sadeghi, A., Schneider, T.: Efficient secure two-party computation with untrusted hardware tokens (full version). In: Sadeghi, A.R., Naccache, D. (eds.) Towards Hardware-Intrinsic Security - Foundations and Practice, pp. 367–386. Springer, Heidelberg (2010)
23. Järvinen, K., Kolesnikov, V., Sadeghi, A.-R., Schneider, T.: Embedded SFE: offloading server and network using hardware tokens. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 207–221. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14577-3_17
24. Katz, J.: Universally composable multi-party computation using tamper-proof hardware. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 115–128. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72540-4_7
25. Koçabas, Ü., Sadeghi, A.R., Wachsmann, C., Schulz, S.: Poster: practical embedded remote attestation using physically unclonable functions. In: ACM Conference on Computer and Communications Security, pp. 797–800 (2011)
26. Kolesnikov, V.: Truly efficient string oblivious transfer using resettable tamper-proof tokens. In: Micciancio, D. (ed.) TCC 2010. LNCS, vol. 5978, pp. 327–342. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11799-2_20
27. Ostrovsky, R., Scafuro, A., Visconti, I., Wadia, A.: Universally composable secure computation with (malicious) physically uncloneable functions. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 702–718. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38348-9_41
28. Pappu, R.S., Recht, B., Taylor, J., Gershenfeld, N.: Physical one-way functions. Science **297**, 2026–2030 (2002)
29. Pappu, R.S.: Physical one-way functions. Ph.D. thesis. MIT (2001)
30. Rührmair, U.: On the security of PUF protocols under bad PUFs and PUFs-inside-PUFs attacks. Cryptology ePrint Archive, Report 2016/322 (2016). http://eprint.iacr.org/
31. Sadeghi, A.R., Visconti, I., Wachsmann, C.: Enhancing RFID security and privacy by physically unclonable functions. In: Sadeghi, A.R., Naccache, D. (eds.) Towards Hardware-Intrinsic Security. Information Security and Cryptography, pp. 281–305. Springer, Heidelberg (2010)
32. Sadeghi, A.R., Visconti, I., Wachsmann, C.: PUF-enhanced RFID security and privacy. In: Workshop on Secure Component and System Identification (SECSI) (2010)
33. Standaert, F.-X., Malkin, T.G., Yung, M.: Does physical security of cryptographic devices need a formal study? (Invited talk). In: Safavi-Naini, R. (ed.) ICITS 2008. LNCS, vol. 5155, p. 70. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85093-9_7

34. Ta-Shma, A., Umans, C.: Better condensers and new extractors from Parvaresh-Vardy codes. In: Proceedings of the 27th Conference on Computational Complexity, CCC 2012, Porto, Portugal, 26–29 June 2012, pp. 309–315. IEEE (2012)
35. Tuyls, P., Batina, L.: RFID-tags for anti-counterfeiting. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 115–131. Springer, Heidelberg (2006). doi:10.1007/11605805_8
36. Vadhan, S.P.: Constructing locally computable extractors and cryptosystems in the bounded-storage model. J. Cryptol. **17**(1), 43–77 (2004)
37. Wolf, S., Wullschleger, J.: Oblivious transfer is symmetric. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 222–232. Springer, Heidelberg (2006). doi:10.1007/11761679_14