Possibilities and Impossibilities for PUF Protocols under PUF Re-Use and PUFs-inside-PUFs Attacks

Marten van Dijk [†] and Ulrich Rührmair [‡]

[†]University of Connecticut, Storrs, CT, USA
[‡]Horst Görtz Institute for IT-Security, Bochum, Germany
`vandijk@engr.uconn.edu,ruehrmair@ilo.de`

Abstract. We continue investigations on the use of so-called Strong PUFs as a cryptographic primitive in realistic attack models. We focus on two scenarios: Firstly, the *PUF re-use model*, where PUFs are being used in more than one protocol run, implying that adversaries may gain *retrospective access* to PUFs after the completion of an earlier protocol. Such re-use appears indispensible in economically and practically viable PUF-applications, but can lead to unexpected attacks. Secondly, the *bad* (or *malicious*) *PUF model*, in which the employed PUFs have been manipulated to possess additional, hidden properties, which enable adversaries to cheat. For the first time, we consider adversaries who build new, malicious PUFs by using old, possibly benign PUFs as subparts of the new PUF, naming this scenario the *PUFs-inside-PUFs model*. In these three models, we obtain and/or formally prove the following results:

– PUF RE-USE, PART I: Any PUF that is *"ideal"* and retrospectively accessible can be replaced by a standard random oracle. By applying the famous Impagliazzo-Rudich result [20], this means that the power of plain Strong PUFs under retrospective access is severely limited; for example, it does not suffice to implement key exchange (KE) or oblivious transfer (OT).
– PUF RE-USE, PART II: Any PUF that is both bad/malicious and retrospectively accessible can be completely eliminated from the protocol. The protocol can be compiled into an information-theoretically equivalent one without this PUF.
– BAD PUFS AND SIMPLIFICATION: As a minor contribution, we simplify a recent OT-protocol for malicious PUFs by Dachman-Soled et al. [4] from CRYPTO 2014. We can achieve the same security properties under the same assumptions, but use only one PUF instead of two.
– PUFS-INSIDE-PUFS, PART I: We propose the new, realistic adversarial model of *PUFs-inside-PUF attacks*, and show that the earlier protocol of Dachman-Soled et al. [4] is vulnerable against PUFs-inside-PUFs attacks (which lie outside the original framework of [4]).
– PUFS-INSIDE-PUFS, PART II: We construct a new PUF-based OT-protocol, which is secure against PUFs-inside-PUFs attacks if the used bad PUFs are stateless. Our protocol introduces the technique of interleaved challenges. We illustrate why the use of interactive hashing in the protocol is necessary, and why a first protocol attempt without interactive hashing fails.

Our findings are not restricted to the UC-framework or its peculiarities, but also apply in stand-alone settings. They have immediate relevance for PUF hardware design: The secure re-use of PUFs in protocols like KE or OT requires properties beyond unpredictability and unclonability. These include *reconfigurability* or *erasability* as well as *certifiability*. Their effective implementation is posed as a central new design goal by our work. Finally, we stress that the secure use of Strong PUFs in standard PUF-based identification protocols remains mostly unaffected by our results, and by attacks in the bad PUF model and the PUF re-use model alike.

1 Introduction and Overview

1.1 Physical Unclonable Functions (PUFs)

Physical unclonable functions (PUFs) [31, 30, 11] are a relatively young cryptographic and security primitive, which has received ever increasing attention from both communities over the last years. A PUF is a randomly structured physical system that can be excited with external physical stimuli, which are called *"challenges"* in the language of the field. Upon application of a challenge c_i, the PUF reacts by producing a corresponding *"response"* r_i. The tuples (c_i, r_i) are commonly termed the challenge-response pairs (CRPs) of the PUF. Each response thereby depends on the applied challenge, and on the unique internal structure of the PUF that is caused by random manufacturing variations. Usually, these variations are assumed to be non-recreatable, and beyond the control of the PUF's own manufacturer. Meanwhile, several technological proposals for realizing PUFs exist (see [25, 36, 37] for an overview), and PUF may become widely available soon; in fact, the commercial usage of PUFs has witnessed a number of breakthroughs recently.

It is essential to distinguish two different PUF types in our context: So-called *Weak PUFs* and *Strong PUFs* [43, 36, 37, 35]. *Weak PUFs* have a small number of challenges (often only one single challenge per PUF). They are usually employed for key storage applications: A single, internal, secret key is derived from the responses of several Weak PUFs. One typical example here are SRAM PUFs [16, 19]: Their random, but reproducible power-up states can be employed to derive a secret key individual to each device [16, 19]. This presupposes that the Weak PUF's responses remain internal, and are non-accessible from the outside. At least in part, this assumption has been put into question in a number of recent invasive attacks on Weak PUFs [18].

So-called *Strong PUFs* follow a different route: They have a very large number of possible challenges, ideally exponentially many in some system parameter (such as in the bitlength of the applied challenges). Their challenge-response mechanism is publicly accessible, meaning that everyone holding the Strong PUF can apply arbitrary challenges and collect corresponding responses. Since there are too many challenges, it is impossible to read out all CRPs of the Strong PUFs, despite the publicly accessible CRP-interface. Furthermore, the internal, response-generating mechanism of Strong PUFs shall be complex: It shall be impossible for the adversary to predict unknown CRPs from a set of known CRPs. Meeting the latter assumption can be difficult in practice, as so-called modeling attacks on Strong PUFs have shown in recent years [43]. Both PUF types have in common that they cannot be physically cloned with perfect precision by existing fabrication technology, hence their name. Recent work has shown that perfect, atomic-scale cloning is not always necessary to mount successful attacks, however: It suffices if the cloning is accurate enough to induce the same challenge-response behavior [18]. Furthermore, certain attacks like PUF-modeling [43] do not even attempt to physically clone the PUF, but try to derive a *digital algorithm* which emulates the PUF's behavior and predicts its responses.

Putting this in a nutshell, Weak PUFs are essentially a new form of a secure key storage element, while Strong PUFs are comparable to a new type of pseudo-random function or hash function, which is derived from the internal physical characteristics of a hardware. [1]

PUF-responses are usually noisy, being affected by temperature variations, voltage fluctuations, aging, measurement inaccuracies, and other effects. For this reason, the responses

[1] For this very reason, Strong PUFs had also been referred to as *"physical random functions"* in early works [11].

of Weak PUFs and Strong PUFs usually must be error corrected. Various techniques have been proposed to this end. Currently, the most popular approach works as follows: Given a response measurement r_i^{Noise} on a given challenge c_i, a noise-free response r_i and some helper data HD are being derived. This helper data can accompany the challenge c_i henceforth. Seeing HD alone shall tell an adversary nothing about the response. But given any noisy response measurement $(r_i^{\mathsf{Noise}})'$ on the challenge c_i (which might be subject to different noise than the original noisy value r_i^{Noise}), the helper data enables derivation of the same noise-free response r_i. Obviously, any such constructions must assume that the noise in the PUF-responses is bounded. In other words: Physical random number generators, which solely output completely random and "noisy" values, are no PUFs.

1.2 Strong PUFs as a Cryptographic Primitive (and a Brief History Thereof)

Initially, PUFs had mostly been considered for simple applications like secret key storage in vulnerable hardware or the identification/authentication of systems [24, 11, 31]. Over the last years, however, the utilization of *Strong PUFs* as a powerful primitive in protocols like KE, OT and BC has increasingly moved into focus.

The basic idea here is relatively simple: In a PUF-protocol with k players $P_1, \ldots, P_k$, one or more Strong PUFs $\mathsf{PUF}_1, \ldots, \mathsf{PUF}_\mathsf{m}$ are being physically transfered between the players. Since the Strong PUFs have a publicly accessible CRP interface, any player P_k who physically holds a PUF at a certain point in time can apply arbitrary challenges c_i to it and collect corresponding responses r_i. As soon as the Strong PUF leaves the hands of P_k, however, he cannot measure new CRPs any more. He will thus find it hard to present or derive the correct response r_i to a previously unknown, randomly chosen challenge c_i. The reasons for this are that (i) the Strong PUF has too many challenges to read out all of them; (ii) the Strong PUF is physically unclonable; and (iii) it is impossible to numerically derive unknown Strong PUF CRPs from known CRPs, at least if the Strong PUF is secure. External adversaries are usually assumed to have access to a PUF while this PUF is in physical transit from one player to another (but there are certain refinements of this basic attack scenario, such as bad PUFs and communicating PUFs, see Sections 1.3 and 5 and [8, 40]).

We stress that *all* recent works on the use of PUFs in advanced cryptographic protocols [7, 33, 46, 34, 1, 28, 8, 29, 40, 6, 4] relate to *Strong PUFs* (even though this is often not being made explicit). Weak PUFs are not useful in scenarios like the above: First of all, their responses are not publicly accessible, and therefore cannot be read out by all the honest protocol participants when the PUF is being passed around. If, on the other hand, the Weak PUF responses were actually made publicly accessible, all their CRPs could be read out too quickly, as each Weak PUF has only one or few CRPs. All of the persons who once had access to the Weak PUF would then know the entire CRP-space, which would disable secure protocols. This means that only the combination of a publicly accessible CRP interface, a huge number of challenges, and the unpredictability of unknown CRPs from known CRPs (i.e., in one word, the use of a *Strong PUF*) makes PUFs applicable as a cryptographic primitive in the sense of the earlier works of [7, 33, 46, 34, 1, 28, 8, 29, 40, 6, 4].

It seems ineffective at first sight that the PUFs need to be transferred *physically* from one player to another. However, there are many scenarios in which this is practical: Consider a bank card containing a PUF, which is being carried by the customer from terminal to terminal, all of the terminals being connected to some central server at the bank. Before the card was issued, the bank has collected many CRPs, which are now stored at the server. This implies that the terminals, the bank server, and partly even the bank card itself, may act as the parties $P_1, \ldots, P_k$ in the abovementioned setting. Something similar applies in situations where some hardware (including computers, laptops, smartphones, etc.) includes a PUF, and is being shipped from its manufacturer (who has collected a large CRP list) to its future owner. The manufacturer and the owner may act as two parties P_1 and P_2 in the above type of setting, with exactly one physical PUF transfer occuring between them.

One key observation is that if a secure Strong PUF is being used in the above scenario, then all players and all adversaries alike can only have partial knowledge about its huge CRP-space, and only one of them can physically hold the (unclonable) Strong PUF at any given point in time. The past few years have shown that a surprisingly large number of protocols can be built on this simple fact, including key exchange [7, 33, 1], olivious transfer [34, 1, 39], or bit commitment [1, 39].

In order to familiarize readers with this line of thinking, we give a simplified precursor of a Strong PUF based key exchange protocol below (compare [7, 33, 46, 1]). The protocol assumes a binary authenticated channel between Alice and Bob (please not that this is a minimal requirement to avoid man-in-the-middle attacks). It assumes that Eve will not manipulate or exchange the PUF while it is in transit, but will merely measure and inspect it arbitrarily, and is designed for a stand-alone setting.

Protocol 1 (Simple Cryptographic Key Exchange with Strong PUFs [7, 33, 46, 1])

1. Alice creates a Strong PUF.
2. She applies randomly chosen challenges $c_1, \ldots, c_k$ to it, and collects the corresponding responses $r_1, \ldots, r_k$.
3. She derives a key K from the responses $r_1, \ldots, r_k$ (possibly applying some error correction).
4. The PUF is being physically transfered to Bob.
5. Bob reports to Alice over the authenticated binary channel that he has received the PUF.
6. Only then, Alice sends the challenges $c_1, \ldots, c_k$ to Bob.
7. Bob applies these challenges to the PUF, and gets (possibly noisy versions of) the same responses $r_1, \ldots, r_k$. He derives the same key K as Alice from these responses (again possibly using error correction).

Protocol 1 nicely illustrates how and why Strong PUF protocols work: While the PUF is in transit, Eve may access it, but she does not know the challenges $c_1, \ldots, c_k$ that Alice used. If the used PUF is a secure Strong PUF, Alice cannot read out its entire CRP-space, since there are too many possible challenges. She also cannot build a digital model that would later allow numeric prediction of the responses $r_1, \ldots, r_k$, and she cannot physically clone the PUF. This means that Eve will always only have partial knowledge about Strong PUF's CRP space. As soon as she learns the challenges $c_1, \ldots, c_k$ by eavesdropping Alice's message, she does not have physical access to the PUF any more. In sum, she will be unable to derive the key K.

We stress again that a Weak PUF cannot be used in the protocol, for the reasons that were given above: First of all, the responses of Weak PUFS are usually kept internal and would not be publicly accessible for Alice and Bob. Secondly, and perhaps yet more importantly, a Weak PUF would only have few CRPs. Even if the responses would be made accessible, Eve could read out *all* CRPs while the Weak PUF is in transit, and thus break the protocol.

The next paragraph briefly summarizes the historic roots of the use of Strong PUFs as a cryptographic primitive. Readers who are not interested may directly skip to Section 1.3.

Brief Historic Account. The full history of Strong PUFs as a cryptographic primitive is quite diverse, with publications spread over a unusually broad range of media. The first PUF-based two-party protocol was published by Pappu in his seminal PhD thesis in 2001 [30]. He suggested the use of a PUF-variant called *"Physical One-Way Functions (POWFs)"* in order to realize bit commitment (BC). His protocol uses the non-invertibility of POWFs, not the unpredictability of standard Strong PUFs, though. In 2004, van Dijk gave an early precursor key exchange (KE) protocol in a patent writing [7]. In 2006, Rührmair gave an independent KE protocol in an internal draft shared with a number of colleagues [33], which was published only later on ePrint [33]. Its protocol is the first to contain steps for authenticating the PUF, and for securing it against being exchanged against another PUF while it is in transit. In a book chapter from 2007, Tuyls and Skoric gave a combined protocol for PUF-based key exchange and authentication in a bank-card scenario [46]. It is arguably the first complex Strong PUF protocol, involving a considerable number of communication steps. It also the first hybrid PUF protocol, where classical computational assumptions and Strong PUF properties are being mixed.

Unfortunately, all of these protocols did not create too much resonance at their time within the community. The interest in Strong PUFs as a cryptographic tool was then arguably sparked a few years later in 2010 by the first published PUF-based oblivious transfer (OT) protocol [34]. The fact that OT was implementable via Strong PUFs immediately allowed realization of BC, KE, and any secure two-party computation, via some well-known reduction results [21]. Also in 2010, the first formal security proof for a Strong PUF protocol (namely the CRP-based identification protocol of Pappu et al. [30, 31]) was being led [35]. In 2011, these two works were soon followed-up by a formal treatment of Strong PUF-based KE, OT and BC in the universal composition framework [1], which was published at CRYPTO 2011. It reached the core cryptographic community, and created substantial follow-up works within it. Many of these dealt with the actual practical security of Strong PUF protocols in realistic attack models [28, 29, 8, 40, 6, 4] (see next Section 1.3).

1.3 Practical Security of Strong PUF Protocols, and Two Attack Models: PUF Re-Use and Bad/Malicious PUFs

Relatively soon after the abovementioned KE, OT, and BC protocols were published [7, 33, 46, 34, 1], a closer look was taken at their practical security in several realistic attack models [42, 8, 28, 29, 38–40]. This led to three different strands of research.

Quadratic Protocol Attacks and Their Relevance. First of all, in a relatively small vein that is often overlooked, a quadratic attack on the security of the OT and BC protocol from CRYPTO 2011 [1] was suggested at CHES 2012 [38] and in JCEN 2013 [39]. It has the effect that an adversary does not need to read out the entire CRP-space of the PUF in order to cheat: If the CRP-space contains n CRPs, then reading out $\sqrt{n}$ of them suffices to break the protocol. Among other things, this has two hardware-relevant effects: (i) The OT- and BC-protocols of [1] cannot be used securely in connection with Pappu et al.'s optical PUF in practice. In order to withstand the quadratic attack, the optical PUF would have to be made extremely large to artificially increase its challenge space, too large to even fit onto a bankcard. (ii) The protocols of [1] cannot be used in connection with electrical Strong

PUFs with bitlength 64, since their challenge space of 2^{64} is reduced to a mere 2^{32} by the attack.

The basic technique underlying the attack is that a cheating Receiver may influence the challenges that an honest Sender applies to the PUF in the course of the protocol [1]. Interestingly, this is also relevant in our context: We show in Section 5.4.1 and in Appendix A that this observation enables an unexpected PUFs-inside-PUFs attack on a first version of a OT-protocol by us that was meant to withstand PUFs-inside-PUFs attacks and stateless bad PUFs. The above technique allws a cheating Receiver to influence the format of some of the challenges that are applied to a PUF-inside-PUF in the course of the protocol. This allows him to covertly communicate some information to the PUF-inside-PUF, and to trigger a special type of malbehavior in the PUF exactly upon these challenges (see Section 5.4.1 and Appendix A).

PUF Re-Use Model and Retrospective Access. Secondly, the abovementioned early protocol suggestions often did not take into account that a PUF would usually be re-used in multiple protocol runs in typical commercial scenarios, and might be physically transfered multiple times between the protocol participants in this context (compare Section 1.2). As an illustrative example, take the abovementioned standard usage example of a PUF on a bank card. Each of the terminals into which the card is inserted might be controlled by Eve. This may obviously provide Eve with additional, retrospective physical access to the PUF after the completion of earlier protocol runs. Another illustrative example in the setting of Section 1.2 is a PUF that is used between Alice and Bob for a key exchange (KE) protocol. The PUF is transfered from Alice to Bob in the course of this protocol. Subsequently, Bob may transfer the same PUF to Claire in another KE scheme. Following the standard adversarial PUF model, Eve will then have physical access to the PUF not only while it is in transit from Alice to Bob, but also while it is transfered from Bob to Claire. I.e., she will have retrospective physical access to the PUF after completion of the first KE between Alice and Bob. These and similar observations necessitate the so-called *"PUF re-use model"* [8, 40], which we sometimes also term *"retrospective access model"* in this paper.

We would like to stress that retrospective access does not only occur in the case of multiple physical transfers of the PUF. If the PUF is part of a complex embedding hardware, like a computer, laptop or smartphone, also viruses or other malware may access the Strong PUF and its interface. Among other things, they may collect CRPs after the completion of an earlier protocol and transfer these to the adversary. Note that by the standard adversarial model for Strong PUFs, their CRP-interface is not protected against adversarial access — if it was, i.e., if we assumed that there are some protected regions of the hardware that cannot be accessed by an adversary, then we could just as well store a classical key in these regions straight away. This observation is particularly relevant for one-transfer protocols like those of CRYPTO 2011 [1].

For all of these reasons, the assumption that *no* retrospective access can occur would be equivalent to supposing that every PUF must be destroyed (or locked away for good) immediately after its first-time use in a single protocol session. It must not be used or accessed any single time afterwards. It is obvious that this assumption would lead to an absolutely uneconomic and unrealistic PUF usage. The only alternative is, however, to assume that adversaries might gain retrospective access, as we do in this paper, and as has been done in earlier works [42, 40].

Bad/Malicious PUFs. Thirdly, recall that a PUF is a piece of hardware that is being physically transfered between the protocol participants. Who can guarantee that this piece of hardware has the expected properties, and only these? Who can guarentee that the party who manufactured the PUF or introduced it to the protocol did not program any covert, malicious behavior into the PUF? Who can guarantee that while the PUF was being passed around from one party to the other, such malicious features where not added afterwards?

Currently, no practically effective methods are known to ensure that a given PUF is not a so-called *"bad"* or *"malicious" PUF*, which has hidden properties that deviate from the expected features of an ordinary PUF [8, 28, 29, 40].[2]

These hidden properties may potentially cover a wide range of unexpected features (compare [40]), including:

– Bad PUFs that are nothing else than digital circuits which generate their responses via a pseudorandom function F which is known to their creator. The creator can hence digitally simulate and predict any responses without having physical access to the PUF, allowing him to cheat in many protocols. These bad PUFs have been termed *"simulatable PUFs"* in [40].
– Stateful bad PUFs that record all challenges that have been applied to them during the course of a protocol, storing these in a special part of their memory that is only known or accessible to their creator. Such PUFs have been named *"challenge-logging PUFs"* in [40].
– Bad PUFs that change their behavior upon a certain triggering signal, which may be one or more specific challenges that are applied to the PUF.
– Bad PUFs that communicate information to other bad PUFs or to their creator when the opportunity arises, and/or which adapt their challenge-response behavior after such communication. These have been denoted *"communicating PUFs"* in [40]. Note that communicating PUFs do not need to be stateful.

We stress that certain bad PUFs are more difficult to implement than others; the used bad PUF type determines how simple and efficient a given attack will be. Communicating PUFs may be the most complex and effortful to realize, but they are also the most effective. We stress that also in the case of classical hardware tokens, equivalent attacks with communicating tokens have been considered recently [9], meaning that it is wise to take the possibility communicating PUFs into account. Especially if a PUF is used in a complex application environment, or within a complex PUF-embedding hardware that is online in some network, communication of the bad PUF (e.g., a normal PUF with a surrounding extra code that runs the communication) with the outside world can indeed be very difficult to prevent.

The bottomline of recent PUF research [8, 40, 28, 29, 42] is therefore that a PUF is not solely an abstract mathematical function that is "transfered" between protocol participants in the Platonic world. Rather, it is a real physical object, which may (or may not!) have the expected properties. In opposition to optical PUFs, electrical PUFs, which mostly are about the size of a large grain of dust, inevitably must be embedded into a larger piece of hardware to be usable and transferrable between users. This embedding hardware again may, or may not, have the expected properties. Furthermore, the economically imperative re-use of PUFs in more than one protocol session may allow adversaries to physically inspect a

[2] The term *"malicious"* PUF has been introduced by Ostrovksy et al. in [28, 29]. Independent work arising around the same time coined the name *"bad"* PUFs [8, 40]. We will use both terms interchangeably, perhaps slightly more often the shorter expression *bad PUFs* for convenience.

PUF more than once. Ideal PUF protocol design should take all the resulting attack surfaces into account, aiming to establish practical security under a minimal set of assumptions.

1.4 Our Contributions

We present five main results in this paper, which concern PUF-protocols in the PUF re-use model/retrospective access model and in the bad/malicious PUF model alike. We describe our contributions below in a self-contained way for readers who do not want to cover the entire paper.

1. Our first result *(Sections 2 and 3)* concerns Strong PUFs protocols in the *"PUF re-use model"* or *"retrospective access model"*. We consider a semi-honest (also known as *"honest-but-curious"*) adversarial behavior in this model, in which adversaries
   - follow the protocol,
   - have so-called *"retrospective access"* to all used PUFs after the end of the protocol, where they may interrogate the PUF with arbitrary challenges,
   - want to derive confidential information after the protocol run (for example both secrets s_0 and s_1 in an OT-protocol).

   We show that in this situation, every PUF that implements a function F can be replaced by an oracle for F, which is accessible for all protocol participants all the time.
   Our above reduction result has immediate consequences for the question whether certain cryptographic tasks can be implemented by Strong PUFs in the retrospective access model *at all.* To illustrate this, we consider an idealized, "most powerful" Strong PUF type. These *"ideal"* PUFs
   - implement a completely random function $f : \{0,1\}^\lambda \to \{0,1\}^\lambda$,
   - have a CRP-behavior that remains unaltered during their entire lifetime,
   - are *"benign"*, i.e., they have no hidden, fraudulent properties that are unknown to the honest parties.

   Our above reduction tells us that any ideal PUF can be replaced by a random oracle in the semi-honest retrospective access model. However, it is long known that the cryptographic power of random oracles is provably delimited via the well-known Impagliazzo-Rudich result (IR) [20], which states that the construction of secure OT or KE on the basis of random oracles alone is practically infeasible. Applying this to our situation, we obtain that the construction of secure OT or KE protocols that solely rest on ideal, retrospectively accessible Strong PUFs is infeasible, too. Let us stress again that the re-use of PUFs in more than one protocol appears economically imperative, making the above infeasibility result highly relevant for practical PUF applications.
2. Our second main finding *(Sections 2 and 4)* treats a combination of the bad PUF and the retrospective access model, i.e., it concerns the use of bad/malicious PUFs to which the adversary also has retrospective access after protocol completion. We show that any PUF which is at the same time
   - simulatable, i.e., the adversary knows a simulation algorithm by which he can numerically predict all responses of the PUF, even without having physical access to the PUF,
   - challenge-logging, i.e., it records all challenges that are applied to it during its lifetime, and the adversary can learn these challenges when he later has physical access to the PUF,

– and retrospectively accessible, i.e., the adversary can access the PUF after protocol completion,

can be *completely eliminated* from any PUF-based protocol. An information-theoretically equivalent protocol can be constructed that runs entirely without that PUF. This finding is yet stronger than the above impossibility results for KE and OT: It shows that bad, retrospectively accessible PUFs are entirely useless in any type of cryptographic protocol, not just in the particular cases of OT or KE.

Our compilation approach extends previously existing impossibility theorems [29, 4] in several aspects: First of all, it also excludes the existence of protocols with computational security, not just of information-theoretic ones. Secondly, it is not limited to specific protocols like OT or KE. Thirdly, it is not limited to specific peculiarities of the UC-model [28], but even works for stand-alone protocols.

3. In a minor contribution asides *(Section 5.1)*, we simplify an existing PUF-based OT-protocol by Dachman-Soled et al. [4] from Crypto 2014. Their protocol is intended to remain secure under stateless malicious PUFs, and uses two PUFs, one $\mathsf{PUF_S}$ introduced by the OT-Sender, one other $\mathsf{PUF_R}$ introduced by the OT-Receiver. The same security properties can also be realized by a simplified protocol: It uses only one $\mathsf{PUF_S}$ introduced by the Sender instead of the two $\mathsf{PUF_R}$ and $\mathsf{PUF_S}$, and thus saves one physical PUF transfer of $\mathsf{PUF_R}$ from the Receiver to the Sender. Our new protocol has similar security guarantees, but also similar vulnerabilities in the so-called *"PUFs-inside-PUFs model"* as the original by Dachman-Soled et al. [4], see below.
4. Fourthly *(Section 5.2)*, we introduce two new, realistic attack models for Strong PUF protocols, that had been overlooked in previous works: *"PUF-modifications"* and *"PUFs-inside-PUFs attacks"*. In PUF-modification attacks, adversaries do not only completely exchange a given PUF against another, bad PUFs, but rather modify a given PUF, and continue to use the modified version. PUFs-inside-PUFs are a special case of PUF-modifications, in which given PUFs (that were initially fabricated by honest parties) are used *inside* or *as subparts of* new, bad/malicous PUFs. For example, a new bad PUF $\mathsf{PUF_{new}}$ could encapsulate an old, previously existing $\mathsf{PUF_{old}}$, and might redirect some of the challenges it receives to $\mathsf{PUF_{old}}$ (that it now encapsulates), and some of them to an internal pseudo-random function PRF that is numerically predictable by the adversary. The outputs of $\mathsf{PUF_{new}}$ would then sometimes be equal to those of $\mathsf{PUF_{old}}$, sometimes to the pseudo-random function PRF. We argue why we consider these two models practically relevant, pointing out that equal assumptions on *"tokens-inside-tokens"* have been considered realistic in the standard literature on hardware tokens [15].

   We then *(Section 5.3)* discuss the effect of "PUFs-inside-PUFs" attacks on the effectiveness of CRP-based PUF-authentication, in particular on the security of the abovementioned OT-protocol by Dachman-Soled et al. [4]. We show that the often suggested "classical" CRP-based PUF-authentication technique [1, 4], in which a set of prerecorded challenges is applied in direct sequence or "blockwise" to the PUF, cannot differentiate with certainty between (i) the abovementioned PUFs-inside-PUF type $\mathsf{PUF_{new}}$ containing $\mathsf{PUF_{old}}$, and (ii) $\mathsf{PUF_{old}}$ itself. An adversary might hence unnoticedly substitute $\mathsf{PUF_{new}}$ for $\mathsf{PUF_{old}}$ in certain protocols and cheat with non-negligible probability. This holds even under the assumption that $\mathsf{PUF_{new}}$ must be stateless.

   This has direct consequences for the OT-protocol by Dachman-Soled et al. [4]: We show that it is vulnerable to stateless PUFs-inside-PUFs attacks. This holds even if a blockwise CRP-based authentication step is applied, as suggested in [4] *(Section 5.3.3)*.

5. Our fifth main result *(Sections 5.4 and 5.5)* lies in developing a secure PUF-based OT protocol for stateless malicious PUFs, which is optimal in several aspects within the given limits of the known impossibility results in this paper and elsewhere [8, 29, 4].
   Our protocol uses only one $\mathsf{PUF_S}$, which is transfered twice (once between the Sender and the Receiver and then back from the Receiver to the Sender). The protocol upholds its security if $\mathsf{PUF_S}$ is malicious, and even if the Receiver modifies $\mathsf{PUF_S}$ into a *stateful* malicious $\mathsf{PUF}^*_\mathsf{S}$ before he returns it to the Sender. Its only assumptions on $\mathsf{PUF_S}$ is that it is stateless when it leaves the hands of the Sender, and that it does not communicate during or after the protocol with adversaries. These are two comparatively mild assumptions. The protocol is optimal in the number of used PUFs, the number of PUF transfers, and the limited statelessness assumption on the used $\mathsf{PUF_S}$. One key technique of our new protocol are *"interleaved challenges"*, i.e., to randomly mix the challenges used to authenticate the PUFs with other challenges that are applied during the protocol.
   We also show asides for didactic purposes *(Appendix A)* that a first protocol attempt, where randomly interleaved challenges are combined in a straightforward fashion with methods from standard PUF-based OT, fails for very subtle reasons. We thereby introduce a new attack strategy where the adversary chooses PUF-challenges maliciously in order to communicate information to a bad PUF.

1.5 Related Work

This work here is a major revision and extension of the work by van Dijk and Rührmair [8], which has been published on eprint in 2012. There is an interesting body of related work concerning the infeasibility of designing higher-level cryptographic protocols based on classical hardware tokens or on PUFs, some of which has appered before, some after [8].

Goyal et al. [15] give an impossibility result for constructing OT protocols based on (classical) stateless hardware tokens. Roughly, they consider curious extension of semi-honest parties, which follow the protocol honestly but may query the token more often than originally intended. In a sense, such curious extensions are thus related to our retrospectively accessible PUFs, where the adversary can query the PUF *after* the actual protocol between honest parties execution ends. In contrast, in the model of Goyal et al. [15] assumes extra queries during the execution of the curious extensions of the honest parties.

The results by Goyal et al. [15] are for stateless hardware tokens different from PUFs. In fact, according to their result information-theoretically secure OT is not possible with their type of tokens, while it can be accomplished using strong PUFs [34, 1]. As our results indicate, switching from genuine PUFs as in [1] to the retrospective access model makes such constructions unlikely, though. Also, instead of ruling out protocols directly in the information-theoretic setting, we present a compiler for the computational setting.

In the full version [28] of their Eurocrypt paper [29], Ostrovsky et al. discuss the impossibility of universally composable bit commitments based on simulatable PUFs (called fully predictable PUFs in their work), if the UC simulator cannot see the queries of the adversary made to the PUF (called oblivious query model). Their impossibility result, however, seems to crucially depend on the UC model and its peculiarities, whereas our result also holds in the stand-alone setting.

In a recent work, Dachman-Soled et al. [4] showed that unconditionally secure oblivious transfer based on malicious, stateful PUFs is impossible. They give a direct proof based on the techniques of Impagliazzo and Rudich [20]. We note that the work by Dachman-Soled et al. [4] unfortunately missed the earlier work of van Dijk and Rührmair [8]. In this earlier work [8], some of the impossibility results and also of the techniques of Dachman-Soled et al. [4], e.g., the tool of a challenge-logging PUF, the connection to Impagliazzo-Rudich, or the approach of XORing the responses of a benign PUF and a bad PUF in order to form a single, benign PUF, had already appeared earlier.

Our work here expands the results of both [8] and Dachman-Soled et al. [4] in several aspects: Regarding the Re-Use of PUFs, it provides a general compiler, works for computationally secure protocols, and, perhap most importantly, also provides results for benign PUFs, as long as they are retrospectively accessible. Regarding bad/malicious PUFs, we simplify Dachman-Soled at al.'s [4] protocol, and show a new attack on it. We thereby introduce a new, so-called PUFs-inside-PUFs strategy, where new PUFs can be built that use old PUFs as "subparts"; the new PUFs may interrogate the old PUFs and pass their responses to the outside world.

We finally note that, related to our compiler, Haitner et al. [17] as well as Mahmoody et al. [26] show that random oracles can be eliminated in some two-party protocols. Besides considering a different primitive, the main difference to our result here is that they start with a statistically secure protocol Π in the semi-honest model, whereas the protocol in the case of our compiler can be computationally secure, but needs to withstand malicious behavior. Technically, their results are based on the "dependency finder" strategy introduced by Impagliazzo and Rudich [20] to deal with intersecting queries to the random oracle, which in turn requires unbounded parties. Our impossibility result relies on a different approach, using the potential of the bad PUF model and thus allows for computationally bounded parties and additional cryptographic assumptions.

1.6 Organization of this Paper

Sections 2, 3 and 4 deal in sequence with our formal impossibility results: Section 2 provides the necessary formal definitions and concepts; Section 3 proves that no Strong PUF based key exchange protocols can be designed which remain secure under PUF re-use and retrospective access; and Section 4 shows that bad PUFs which are both challenge-logging, simulatable, and can be retrospectively accessed after the end of the protocol, can be fully removed from any protocol.

Section 5 and Appendix A treat the realization of secure Strong PUF based OT-protocols in the face of (stateless) bad/malicious PUFs and PUFs-inside-PUFs attacks: Section 5.1 presents a slight simplification of the original protocol of Dachman-Soled et al. [4]; Section 5.2 introduces the attack models of *"PUF-modification"* and *"PUFs-inside-PUFs"*; Section 5.3 illustrates the problems of CRP-based PUF-authentication in the face of PUFs-inside-PUFs attacks and demonstrates the vulnerability of a protocol by Dachman-Soled et al. from Crypto 2014 against PUFs-inside-PUFs techniques. [3] Section 5.4 presents a countermeasure against PUFs-inside-PUFs attacks named interleaved challenges, and discusses (partly in Appendix A) why a first protocol attempt by us for restoring the security of the protocol by Dachman-Soled et al. [4] fails. Section 5.5 finally presents a PUF-based OT-protocol based on interleaved challenges and interactive hashing that is secure against bad PUFs, PUF-modifications and PUFs-inside-PUFs attacks under somewhat minimal assumptions.

We conclude the paper in Section 6.

[3] We stress again in this context that the attack model of PUFs-inside-PUFs is a realistic scenario in our opinion, but lies outside the original setting of Dachman-Soled et al. Their formal proofs, which rest on a different adversarial model, remain formally correct in their own framework.

2 Preliminaries

2.1 Functional Properties of Physical Unclonable Functions

We define infinite families of PUFs (cf. [1]), as such an asymptotic PUF-treatment is necessary to argue along the lines of secure multi-party computation and of Impagliazzo-Rudich [20].

Definition 1 (PUFs). *A family $\mathcal{P}$ of PUFs consists of a family $\mathcal{G} = (G_\lambda)_\lambda$ of function sets $G_\lambda = \{g\}$ and two procedures, called* `PUFcreate` *and* `PUFeval`*, where* `PUFcreate` *on input pid $\in \{0,1\}^\lambda$ samples a function $g : \mathcal{C}_\lambda \to \mathcal{R}_\lambda$ from G_λ, describing a PUF for identifier pid with challenge space $\mathcal{C}_\lambda$ and response space $\mathcal{R}_\lambda$. The procedure from thereon refers to the function as g_{pid}. When* `PUFeval` *is called for PUF pid and challenge c it returns $r = g_{pid}(c)$. (For sake of convention, if* `PUFcreate` *is called about a previously seen pid, it ignores the request; if* `PUFeval` *is called about a fresh pid it also ignores the request.)*

Definition 1 assumes that suitable error correction techniques have been applied to the PUF, leading to a stable PUF output. Note that the definition can easily be adjusted to the case of noisy PUF outputs by replacing $g_{\text{pid}}(c)$ by a random variable, but we will not follow this route here. Also note that in principle we do not forbid that two functions $g_{\text{pid}}, g_{\text{pid}'}$ for distinct pid $\neq$ pid$'$ are identical. We often simply speak of a PUF instead of a family of PUFs.

In this paper, we exclusively consider PUFs that have challenge and response set $\{0,1\}^\lambda$, with λ being the security parameter. One reason is our focus on ideal PUFs with maximal input and output complexity that we discussed above. But there is another reason: As already observed by Brzuska et al. [1], in order to achieve output length of λ in practice, fuzzy extraction of several consecutive responses can be applied, or the concatenation of the λ responses of λ independent hardware instantiations of a PUF to the same challenge can be used.

We next capture the notion of ideal families of PUFs which basically implement random functions.[4] To express that the PUF looks like a random function we consider a distinguisher D which has oracle access to either the set of all PUFs from the family, or to random functions. Algorithm D should not be able to distinguish the two cases. Below we make the distinguisher unbounded. Still, the number or queries to the oracles is polynomially bounded. Formally, we define $\mathcal{P}^\$$ as the family of PUFs where, upon each creation, we replace the actually sampled function g_{pid} by a truly random function with the same challenge and response space. In other words, an ideal PUF essentially implements a random oracle but enforces restrictions:

Definition 2 (Ideal PUFs). *A family $\mathcal{P}$ of PUFs is called ideal if for all $\lambda \in \mathbb{N}$, each function $g_{pid} \in G_\lambda$ has challenge and response set $\{0,1\}^\lambda$, and for any unbounded algorithm D and any sequence $\{z_\lambda\}_{\lambda \in \mathbb{N}}$, the advantage by which $D^{(\cdot)}(z_\lambda)$ can distinguish with a polynomial number of oracle queries between $\mathcal{P}$ and $\mathcal{P}^\$$, is negligible.*

[4] This is the ideal view of the type of PUFs that has been investigated in most earlier work on PUFs (e.g. [31, 30, 35, 1]): Namely PUFs with a large (for practical purposes exponential) number of CRPs, whose challenge-response interface is accessible by everyone who holds physical possession of the PUF. Such PUFs have sometimes been referred to as Strong PUFs or Physical Random Functions [12, 16, 37].

We next characterize the notion of simulatable PUF for our impossibility result. This basically says that one cannot distinguish *efficiently* between the outputs of a PUF and the ones by a binary algorithm SWsim (for software simulation). Algorithm SWsim is defined to be deterministic and stateless, but it accepts a uniformly chosen key $k_{\text{pid}} \leftarrow \{0,1\}^{\lambda}$ as intrinsic randomness, which can also serve as a seed to generate additional pseudorandomness if necessary. Requiring the existence of algorithm SWsim will later enable the adversary to create a valid-looking and yet predictable hardware token. In addition to the evaluation algorithm, we also assume for formal reasons that there exists a stateful algorithm SWeval, taking a PUF identifier pid and a challenge c, and looking up the matching key k_{pid} to pid and returning $\mathsf{SWsim}(k_{\text{pid}}, c)$. We also introduce an algorithm SWcreate which in input pid internally picks a random key k_{pid}, such that, formally, SWcreate and SWeval provide the same interfaces as `PUFcreate` and `PUFeval`. When an algorithm, as below, has oracle access to both interfaces we simply write SW and $\mathcal{P}$ for the oracles.

Definition 3 (Simulatable PUF). *A family $\mathcal{P}$ of PUFs is called simulatable if there exists a deterministic polynomial-time algorithm SWeval such that for any probabilistic polynomial-time algorithm D and any sequence $\{z_\lambda\}_{\lambda\in\mathbb{N}}$, the advantage by which $D^{(\cdot)}(z_\lambda)$ can distinguish with a polynomial number of oracle queries between oracle SW (as defined above) and oracle $\mathcal{P}$, is negligible.*

Note that ideal PUFs are simulatable if one-way functions exist, because one can simulate random functions via pseudorandom functions, and ideal PUFs are indistinguishable from random functions (even for unbounded distinguishers). If we consider only distinguishers with a fixed, universal polynomial bound of queries, then ideal PUFs are simulatable via poly-wise independent hash functions. This applies in principle to our compiler result in the next section.

2.2 PUF Functionalities for Protocols

Following Brzuska et al. [1] we define PUFs in protocols via functionalities, only that we give reduced and adapted versions, since we do not require the full power of the UC model. In fact we only look at the two-party scenario here, with parties P_1 and P_2. Note that a PUF functionality merely describes the interface for interactions, not the functional properties. The functionality is stateful.

Definition 4 (PUF Functionality). *A PUF functionality $\mathcal{F}[\mathcal{P}]$ for a family of PUFs $\mathcal{P}$ consists of three commands:*

- *When a party P_i calls* `create`*(pid) then the functionality first checks that there is no entry $(pid, *, *, *, *)$ stored yet (else it ignores the request), then calls* `PUFcreate`*(pid) for the identifier pid and stores the tuple $(pid, g_{pid}, g_{pid}, P_i, P_i)$ representing the identifier, the initial function (given implicitly here through* `PUFeval`*$(pid, \cdot)$), the current function (for forward compatibility for the malicious model), the manufacturer's identity, and the party currently in possession of the PUF.* [5]
- *When a party P_j calls* `handover`*(pid) then check for an entry $(pid, g_{pid}, g, P_i, P_j)$ and, if such an entry exists, replace it by $(pid, g, g, P_i, P_{3-j})$ to identify the other party as the new holder. Return pid to P_{3-j}. If no such entry exists, then ignore the request.*

[5] To avoid collisions we assume that pid is prepended by the creator's identity, P_1 or P_2.

– *When a party P_j calls* `evaluate`(pid, c) *then check for an entry* $(pid, g_{pid}, g, P_i, P_j)$ *and, if existent, return* $r = g(c)$*; else ignore the request. For sake of extension we also let the functionality maintain an initially empty list* $L_{\mathcal{F}}[\mathcal{P}]$ *to which* (pid, c, r) *is appended, albeit this is not used here.*

To capture PUF-based protocols we start with the regular notion of a two-party protocol Π between parties P_1 and P_2 and augment it by the PUF functionality above as an oracle available to both parties (see [13]). Similar to the case of oracle Turing machines we can then augment the view of a party, containing its input, randomness, and the sequence of incoming (binary) messages from the other party, by interleaving the input messages with the interactions with the PUF functionality. In this sense the definition of a view of the first or second party in an execution for inputs x and y, denoted $\mathtt{view}_1^{\Pi}(x, y)$ and $\mathtt{view}_2^{\Pi}(x, y)$, respectively, is defined as in the oracle case. Note that the views are random variables, with the probability space consisting of the random choices of the parties and the functionalities randomness.

Transparent access of PUF now says that one party can always determine if the other party evaluates a PUF. A classical example is if the protocol description determines such queries by a fixed structure, and evaluation queries do not occur in dependence of the state. This property is defined for sake of simplicity by having one party inform the other party about such queries, with the intuition that this only formalizes the ability to predict such queries:

Definition 5 (Transparent PUF Evaluation Patterns). *We say that a PUF-based two-party protocol has a* transparent PUF evaluation pattern *if during any protocol execution either party, before calling the PUF functionality about some* `evaluate`(pid, c)*, sends a special digital message "*query pid*" to the other party and in return receives a digital message "*ok*".*

2.3 Security of PUF-based Protocols

We now clarify a few notions that are related to the security of PUF-based protocols. By construction, there is no set-up assumption like a common reference string; we only consider plain two-party protocols (in the presence of PUFs or, more generally, of other oracles).

We next require a definition of semi-honest behavior in the context of PUFs (or other oracles) for our impossibility proofs. In the standard semi-honest model (where the adversary follows the protocol with the exception that it keeps the values of all its intermediate computations), the joint views of the adversary and the honest player must be close to a simulated view in the ideal model. See again [13]. We adopt this definition here, too.

Semi-honest security. We can now define semi-honest behavior in PUF protocols. Basically, as in the classical case we require that simulators S_1 and S_2 (now with access to the PUF family) can generate simulated views which look like the one of the actual parties, but without knowing anything about the other party's input beyond the functional result of the protocol.

We note that there are two options about the distinguisher. Either we give it access to the PUF functionality when inspecting the genuine or simulated view — or we deny the distinguisher access to the PUF. We opt for the latter to strengthen our impossibility results. We therefore say that two random variables $\{Y_{z_\lambda}\}_{\lambda \in \mathbb{N}}$ and $\{Z_{z_\lambda}\}_{\lambda \in \mathbb{N}}$, indexed by a sequence of values z_λ, are statistically (resp. computationally) indistinguishable if for any unbounded (resp. probabilistic polynomial-time) algorithm D the difference

$$|\mathrm{Prob}[\,D(z_\lambda, Y_{z_\lambda}) = 1] - \mathrm{Prob}[\,D(z_\lambda, Z_{z_\lambda}) = 1]|$$

is negligible. We write $\overset{s}{\equiv}$ and $\overset{c}{\approx}$ in this case.

Definition 6 (Security w.r.t. semi-honest behavior). *Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ be a deterministic*[6] *functionality, and denote the first (resp., second) output of $f(x,y)$ by $f_1(x,y)$ (resp., $f_2(x,y)$). Let $\Pi^{\mathcal{O}}$ be a two-party protocol for computing f based on an oracle $\mathcal{O}$. We say that Π computes f computationally secure against semi-honest adversaries relative to $\mathcal{O}$ if there exist probabilistic polynomial-time algorithms $S_1^{\mathcal{O}}$ and $S_2^{\mathcal{O}}$, making at most a polynomial number of oracle queries, such that*

$$\{S_1^{\mathcal{O}}(x, f_1(x,y))\}_{x,y\in\{0,1\}^*} \overset{c}{\approx} \{\mathtt{view}_1^{\Pi^{\mathcal{O}}}(x,y)\}_{x,y\in\{0,1\}^*}$$
$$\{S_2^{\mathcal{O}}(y, f_2(x,y))\}_{x,y\in\{0,1\}^*} \overset{c}{\approx} \{\mathtt{view}_2^{\Pi^{\mathcal{O}}}(x,y)\}_{x,y\in\{0,1\}^*}$$

where $|x| = |y|$.

Note that we easily obtain the statistical case if we allow for unbounded S_1, S_2 (still with polynomial many oracle queries) and consider statistical indistinguishability instead.

Modeling bad and retrospectively accessible PUFs. In the bad PUF model of simulatable and challenge logging PUFs we augment the PUF functionality by two interfaces for the adversary, allowing it to place an arbitrary algorithm A inside the PUF, and to read out previous evaluation queries. Setting the algorithm A we allow this procedure to depend on further PUFs ("embedding PUFs in PUFs"). The adversary has to specify the identifiers $\mathrm{pid}_1, \mathrm{pid}_2, \dots$ of these embedded PUFs and hold them upon initialization. Embedded PUFs (different from the original one) are then no longer in possession of the adversary and cannot be, for example, embedded in another PUF. This ensures unclonability. Note that our model does not allow for nested embeddings, nor for decomposition of assembled PUFs. Neither property is required for our negative result, nor could the adversary in our commitment scheme take advantage of these properties.

Definition 7 (Malicious/Bad PUF Functionality). *A bad PUF functionality $\mathcal{M}[\mathcal{P}]$ for a family of PUFs $\mathcal{P}$ consists of three commands* ***`create`***, ***`handover`***, ***`evaluate`*** *as a regular PUF functionality, but also the two following commands:*

- *When a party P_i calls* ***`set-algo`***$(pid, A, pid_1, pid_2, \dots)$ *for an oracle circuit (description) A and a set of PUF identifiers $pid_1, pid_2, \dots$ then the functionality verifies if there are entries $(pid, g_{pid}, *, *, P_i)$, $(pid_j, *, *, *, P_i)$ and, if so, replaces it by $(pid, g_{pid}, A, P_i, P_i)$ such that subsequent evaluations now compute $A(c)$ instead of $g_{pid}(c)$. For any $pid_j \neq pid$ the functionality replace entries $(pid_j, *, *, *, P_i)$ by $(pid_j, *, *, *, \perp)$.*
- *When a party P_i calls* ***`get-log`***(pid) *then check for an entry $(pid, *, *, P_i, P_i)$ and, if such an entry exists, output the current state of the list of all stored pairs (pid, c, r); recall that storage has been implemented by our definition of regular PUFs. If no such entry exists, then ignore the request.*

[6] Our results also work for randomized functionalities, but deterministic functionalities simplify the presentation.

In the retrospective access model we allow the adversary to make a final reveal of the log of any PUF *manufactured by the adversary*, even if it is currently not in possession of the PUF. In the generally retrospectively accessible model we allow the adversary access to any PUF, even if it has been *created by an honest party*. In both cases access means that the adversary can make any calls about any of the available commands (where we leave it up to the adversary to avoid endless recursive calls):

Definition 8 (Retrospectively accessible PUF Functionality). *A retrospectively accessible (resp. generally retrospectively accessible) PUF functionality $\texttt{retro-}\mathcal{M}[\mathcal{P}]$ for a (possibly malicious) functionality $\mathcal{M}[\mathcal{P}]$ provides the same commands as the PUF functionality, plus one additional command:*

- *In the retrospective access case, if a party P_i calls $\texttt{retro-acc}(pid, \texttt{cmd+arg})$ then check for an entry $(pid, *, *, P_i, *)$ and, if such an entry exists, process the command $\texttt{cmd+arg}$ (with arguments) for the PUF pid and return the reply; if no such entry exists then ignore.*
- *In the general retrospective access case, if a party P_i calls $\texttt{retro-acc}(pid, \texttt{cmd+arg})$ then check for an entry $(pid, *, *, P_j, *)$ and, if such an entry exists, first replace P_j by P_i, then process the command $\texttt{cmd+arg}$ (with arguments) for the PUF pid and return the reply, and finally re-set the manufacturer entry for pid to P_j; if no such entry exists then ignore.*

Partially Malicious Behavior of Adversaries. We next define security against malicious parties P_1^* or P_2^*. We only require security against adversaries using the additional commands of the malicious PUF (or the retrospectively accessible PUF) functionality, and otherwise behaving honestly, e.g., not aborting and not substituting the input value. In particular, in the retrospective access model the party uses the `retro-acc` command only as the last step before stopping. This simplifies the definition below. We call such adversaries *partially malicious*, and denote the views of such partially malicious algorithms by $\texttt{view}^{*\Pi}_{1}(x, y)$ and $\texttt{view}^{*\Pi}_{2}(x, y)$, respectively. We also stipulate that partially malicious parties are polynomial time.

We give the definition only for the malicous PUF model; the retrospective access and general retrospective access cases are defined analogously.

Definition 9 (Security w.r.t. partially malicious behavior in the bad-PUF model). *Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ be a deterministic functionality, and denote the first (resp., second) output of $f(x,y)$ by $f_1(x,y)$ (resp., $f_2(x,y)$). Let $\Pi^{\mathcal{M}[\mathcal{P}]}$ be a two-party protocol for computing f based on a family of bad PUFs $\mathcal{M}[\mathcal{P}]$. We say that Π computes f securely w.r.t. partially malicious behavior in the bad PUF model if for any partially malicious probabilistic polynomial-time algorithms P_1^* and P_2^* there exist probabilistic polynomial-time oracle algorithms $S_1^{\mathcal{M}[\mathcal{P}]}$ and $S_2^{\mathcal{M}[\mathcal{P}]}$ such that*

$$\{S_1^{\mathcal{M}[\mathcal{P}]}(x, f_1(x,y))\}_{x,y\in\{0,1\}^*} \overset{c}{\approx} \{\texttt{view}^{*\Pi}_{1}(x,y)\}_{x,y\in\{0,1\}^*}$$
$$\{S_2^{\mathcal{M}[\mathcal{P}]}(x, f_2(x,y))\}_{x,y\in\{0,1\}^*} \overset{c}{\approx} \{\texttt{view}^{*\Pi}_{2}(x,y)\}_{x,y\in\{0,1\}^*}$$

where $|x| = |y|$.

From Malicious to Partially Malicious Security. Our definition of partially malicious behavior restricts the adversary in the sense that it can only misbehave in a controlled way. At the same time, we also restrict how the simulator in our definition can interact with the ideal functionality f. Namely, according to our approach simulator S_1 receives, in addition to the party's input x, the function output $f_1(x, y)$ for the party. In contrast, in the (fully) malicious model S_1 can make a single, arbitrary query to $f_1(\cdot, y)$ for an input of its choice. Likewise for S_2. This means that partially malicious security may not be implied by (fully) malicious behavior in general, as the fully malicious simulator has more power in principle.

However, the common way to build simulators in proofs for full malicious security is to extract the input x^* inserted by the malicious party into the execution (here: $x^* = x$ by construction of our adversary), and then letting the simulator query $f_1(\cdot, y)$ exactly about this extracted value x^*. In this sense, if such a simulator exists —and we merely need the pure existence for our result to hold— then malicious security does imply partially malicious security.

Moreover, for specific functionalities such as OT with $f_1(x, y) = \epsilon$ and $f_2(x, y) = x_y$ for $x = (x_0, x_1)$ and $y \in \{0, 1\}$, we can actually show that malicious security implies the existence of such a simulator. In this case the simulator S_1 in the fully malicious model does not gain any advantage from the oracle query to $f_1(\cdot, y)$, because the output is constantly ϵ. The simulator $S_2^{\mathcal{M}[\mathcal{P}]}(y)$ in the fully malicious case may make a query to $f_2(x, \cdot)$ for a different value $y^* \neq y$ (with non-negligible probability), but then this simulator must fail for some inputs to output an indistinguishable view from the one of our (partially) malicious adversary. That is, if $x_0 \neq x_1$ then S_2 only learns x_{y^*} from such a query, but our adversary outputs x_y from the interaction with the honest sender. Since the distinguisher receives $x = (x_0, x_1)$ and y as input, it can easily be checked that x_y appears in the view. The simulator S_2, however, must fail to output this unknown value if it queries its ideal functionality with $y^* \neq y$, such that this probability of making such a query cannot be non-negligible for such a simulator. If the probability for such a query is negligible, though, then we can simulate the answer with overwhelming probability if we only receive $f_2(x, y)$ as input, as our partial malicious simulator does.

More abstractly, if there exists an (infinite) sequence of values $x_\lambda, x'_\lambda, y_\lambda$ such that $f_b(x_\lambda, y_\lambda) \neq f_b(x'_\lambda, y_\lambda)$, but for any $y^*_\lambda \neq y_\lambda$ we have $f_b(x_\lambda, y^*_\lambda) = f_b(x'_\lambda, y^*_\lambda)$, then any fully malicious simulator $S_b^{\mathcal{M}[\mathcal{P}]}$ needs to make its only query to f_b about its input value with overwhelming probability and can thus be simulated by our partially malicious simulator, receiving that function value as input. Also, if f_b is constant (as f_1 in case of OT) then our partially malicious simulator can also execute the fully malicious simulator, given $f_b(x, y)$ as input only.

3 Replacing Retrospectively Accessible Ideal PUFs by Random Oracles in PUF-Protocols

In this section, we prove that any retrospectively accessible ideal (and benign) PUF can be replaced by a random oracle in any given PUF protocol. The new protocol is equivalent to the old one in the strongest (i.e., information-theoretic) sense.

There is an immediate implication from our method: If a PUF-protocol uses *only* retrospectively accessible ideal PUFs, then it can be compiled in a protocol that uses *only* random oracles instead. Applying the famous Impagliazzo-Rudich (IR) result [20] to this case, we obtain that constructing provably secure protocols for OT and KE that *solely* use retrospectively accessible ideal PUFs is practically infeasible. More precisely, IR tells us that constructing such a protocol and proving its security would result in a proof that $\mathcal{P} \neq \mathcal{N}P$, which currently is considered infeasible.

Our first observation is that we can safely replace any ideal PUF by a random oracle in the case of partially malicious behavior. The reason is that any restrictions about not accessing the PUF when it is not in the physical possession of a certain party will also be obeyed by all honest protocol participants in the random oracle case.

Lemma 1. *If $\Pi^{\mathcal{P}}$ computes a function f securely w.r.t. partially malicious behavior in the generally retrospectively accessible PUF model for an ideal PUF, then protocol Π^{RO}, where each query $\texttt{PUFeval}(pid, \cdot)$ to a PUF is replaced by a call to a random oracle $\mathsf{RO}(pid, \cdot)$, computes f securely w.r.t. partially malicious behavior in the random oracle model.*

Recall now that Impagliazzo and Rudich [20] show that there is a concrete (outsider) adversary against a key exchange protocol, the IR adversary $\mathcal{A}_{\text{ke}}^{\mathsf{RO}}$, which can compute the secret key with sufficiently high probability (in the presence of a random oracle RO). Here it is important that the adversary $\mathcal{A}_{\text{ke}}^{\mathsf{RO}}$, receiving the transcript of a communication between the two honest parties P_1, P_2, has access to the random oracle; else it is trivial to construct a secure key exchange protocol. We next argue that the IR adversary can be used to break the security of OT protocol in the retrospective access model.

Consider first the construction of key exchange from oblivious transfer (see, e.g., [13]). There, P_1 chooses a random key $K \leftarrow \{0,1\}^\lambda$ and both parties engage in an OT protocol execution with P_1 having input $(K, 0^\lambda)$ and P_2 asking to receive the first value, $b = 0$. The security argument is by two game hops, one time arguing that from the perspective of $\mathcal{A}_{\text{ke}}^{\mathsf{RO}}$ using inputs $(K, 0^\lambda)$ and $b = 1$ instead for the OT protocol is indistinguishable by the receiver's privacy. This means that the IR adversary would still be able to find K with sufficiently high probability. Indistinguishability, too, holds if one uses an independent $K' \leftarrow \{0,1\}^\lambda$ for inputs $(K', 0^\lambda)$ and $b = 1$ instead for the OT protocol, now because of the sender's privacy. But now the communication transcript is independent of the actual key K and $\mathcal{A}_{\text{ke}}^{\mathsf{RO}}$ could only succeed with the guessing probability.

The important observation in the above argument is that the IR adversary $\mathcal{A}_{\text{ke}}^{\mathsf{RO}}$ yields two concrete adversaries $\mathcal{A}_{\text{S-OT}}^{\mathsf{RO}}$ and $\mathcal{A}_{\text{R-OT}}^{\mathsf{RO}}$ against the sender's privacy resp. the receiver's privacy in the OT protocol. This holds for honest executions for which the adversary, receiving the genuine transcripts, breaks privacy of the corresponding party, even in a *game-based indistinguishability notion* by predicting the party's inputs beyond guessing. But we can then use $\mathcal{A}_{\text{S-OT}}^{\mathsf{RO}}$ and $\mathcal{A}_{\text{R-OT}}^{\mathsf{RO}}$ to mount a partially malicious attack in the generally retrospectively accessible PUF model: The derived adversary honestly follows the protocol to generate the communication transcripts and then uses the general retrospective access to the PUF resp. random oracle to run algorithm $\mathcal{A}_{\text{S-OT}}^{\mathsf{RO}}$ resp. $\mathcal{A}_{\text{R-OT}}^{\mathsf{RO}}$ on this transcript. Here we require access to all PUFs, not only those created by a party.

Next observe that a successful adversary against game-based security as above violates simulation-based security in the partially malicious model, even for distinguishers without access to the random oracle. The distinguisher simply checks the prediction of the adversary resp. the simulator against the protocol input. This now contradicts the security of our original protocol for OT in the retrospective access model against partially malicious behavior. The final step is to note that we can implement the IR adversary efficiently if $\mathcal{P} = \mathcal{NP}$. We thus obtain:

Theorem 1. *There is no provably secure two-party PUF-protocol for computing oblivious transfer w.r.t. partially malicious behavior which solely uses ideal PUFs that are retrospectively accessible, provided that $\mathcal{P} = \mathcal{NP}$.*

The same argument of course applies to other advanced protocols like KE, too.

Note again that this implies that any such provably secure protocol entails a proof that $\mathcal{P} \neq \mathcal{NP}$. Giving such a proof is considered to be hard, such that it also follows that devising such a protocol as at least equally hard. As pointed out in [20, 45, 32], if one can embed a PSPACE-complete language into the oracle, then even this complexity-theoretic assumption vanishes completely. It remains arguable if such an embedding works for the case of PUFs, though. On one hand, the associated functions of good PUF should have no structure. On the other hand, the embedding is only known to work for some primitives such as one-way functions; for the random-looking PUF function this may not be admissible.

Furthermore, we remark that the use of ideal PUFs in our proof is no weakening of our results: Recall that any generic PUF-based OT or KE scheme that holds for PUFs in general would even more hold for ideal PUFs with close to perfect randomness, such as the ideal (and benign) PUFs considered by us in this paper.

4 Completely Eliminating Retrospectively Accessible Bad PUFs from PUF-Protocols

In this section, we show something yet stronger than the results of the last section. We prove that any challenge-logging, simulatable, retrospectively accessible PUF can be completely eliminated from a PUF protocol. An information-theoretically equivalent protocol can be constructed that does not use this PUF at all. This means that these PUFs are not useful at all in constructing cryptographic schemes. Saying this in yet other words, if we consider security in a combination of the retrospective access model and the bad PUF model, and if we assume that the adversary is capable of fabricating PUFs that are both simulatable and challenge-logging at the same time, then any such PUF is useless in any cryptographic protocols. Our result does not only hold for the advanced cases of OT and KE, but for any type of cryptographic scheme.

Let SW be the algorithm implementing simulatability. Assume that we have an arbitrary PUF protocol $\Pi^{\text{retro-}\mathcal{M}[\mathcal{P}]}$ which privately computes a function f against partially malicious adversaries in the retrospectively accessible bad PUF model (in the computational sense). In particular, the protocol must be secure against a special adversary $\mathcal{A}_i$ (impersonating party P_i) which we call the *simulating-and-logging* adversary. This adversary acts exactly like party P_i, except for the following steps:

- Whenever P_i would create a genuine PUF, the adversary $\mathcal{A}_i$ also calls `create` but in addition picks a random key $k \leftarrow \{0,1\}^\lambda$ for the algorithm SWsim and immediately places $\mathsf{SWsim}(k, \cdot)$ into the (simulatable) PUF by calling `set-algo`.
- Whenever P_i would receive a PUF back from P_{3-i}, i.e., when a `handover` returns pid to P_i, adversary $\mathcal{A}_i$ uses the challenge-logging mechanism, calling `get-log`(pid), to recover all queries made to the PUF (and thus putting them into its view).
- At the end of a protocol execution, for any PUF pid created by $\mathcal{A}_i$, the adversary finally calls `retro-acc`(pid, `get-log`(pid)) to recover all challenge response pairs stored in the PUF.

The adversary eventual outputs its view. We will next turn the protocol $\Pi^{\textsf{retro-}\mathcal{M}[\mathcal{P}]}$ into a digital-only protocol Π' between parties P_1' and P_2' without relying on the PUF functionality. The derived protocol will be secure in the semi-honest case, but we can then thwart malicious behavior by simply using the GMW compiler [14].

The idea of protocol Π' is as follows. The parties P_1', P_2' essentially behave like $\mathcal{A}_1$ and $\mathcal{A}_2$, except that P_1' instead of placing SW for the key into the PUF token like $\mathcal{A}_1$, simply keeps the key to itself, not sending anything. In order to allow P_2 to evaluate the PUF, we let P_2' send the corresponding challenge to P_1' in clear, and have P_1' return the SW output. At first this seems to violate privacy of the queries. But remember that P_1' learns the queries anyway through the logging, once the PUF returns! If the PUF is not returned then we use the ability of retrospectively accessible case to read out the final queries. Hence, no extra information is leaked to P_1', it may only be revealed earlier now. But since the parties are semi-honest, this is not exploited by P_1'. We use the same argument for P_2'. We next make the transformation and the idea more formal, relying on the transparent PUF evaluation pattern of the protocol.

PUF-Eliminating Compiler. Let $\Pi^{\textsf{retro-}\mathcal{M}[\mathcal{P}]}$ be a simulatable, retrospectively accessible, bad PUF two-party protocol between P_1, P_2 with transparent PUF evaluation patterns. Let SWsim be the algorithm for simulatability of the PUF. Construct the following purely digital protocol Π' between P_1', P_2' from Π, where P_1', P_2' behave like P_1, P_2 —and use these algorithms as sub routines— with the following differences:

– Whenever P_i creates a PUF for pid in the protocol via `create`, then P_i' saves pid and stores it locally. It also picks a key $k_{\text{pid}} \leftarrow \{0,1\}^\lambda$ for algorithm SWsim under the identifier pid.
– If P_i sends a PUF for pid to P_{3-i} in Π via `handover`, then P_i' sends a special handover message with pid to P_{3-i}'.
– If P_i' receives a special handover message with pid, it forwards this to P_i as a result of a `handover` query by P_{3-i}.
– If P_i would query a PUF for pid about a challenge c and the corresponding PUF then P_i' now forwards c and the corresponding PUF identifier to P_{3-i}' and waits to receive a reply r; it forwards r as the PUF's reply to P_i.
– Whenever P_i' receives a challenge query c for some of its PUF identifiers pid, it evaluates $\mathsf{SWsim}(k_{\text{pid}}, \cdot)$ on the corresponding key and the challenge c, and returns the reply r.

Theorem 2. *If the two-party protocol $\Pi^{\textsf{retro-}\mathcal{M}[\mathcal{P}]}$ computes a function f securely w.r.t. partially malicious behavior in the retrospectively accessible bad PUF model, and the PUF is simulatable by SWsim, then the protocol Π' in Construction 4 also computes f securely in the semi-honest model.*

Proof. We need to build simulators S_1' and S_2' which, on inputs $(x, f_1(x,y))$ resp. $(y, f_2(x,y))$, can create indistinguishable views from the ones of P_1' and P_2' in Π' for the same inputs (x,y). We describe first how to build S_1'. Note that, by assumption, we have a simulator $S_1^{\textsf{retro-}\mathcal{M}[\mathcal{P}]}$ for the view of $\mathcal{A}_1$, the simulating-and-logging version of P_1 with the final retrospectively accessible read-out. Simulator S_1' first runs $S_1^{\textsf{retro-}\mathcal{M}[\mathcal{P}]}(x, f_1(x,y))$ to create a view `view` close to the one of $\mathcal{A}_1$ in an protocol execution. For each interaction of the simulator S_1 with the PUF oracle, simulator S_1' instead simulates the PUF via SW as in the simulatability definition 3. Note that all the additional interfaces of the retrospectively accessible bad PUF can be easily simulated, too.

The view eventually output by S_1 for the simulated PUF via SW is indistinguishable from a view generated with access to the original PUF $\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]$ for any distinguisher D. Given such a "simulated" view, which contains all the log entries of any PUF, simulator S_1' uses the transparency property of PUF evaluations, to efficiently "unfold" this view into one for P_1' in Π':

- For each of the log sequences in the view, i.e., when the PUF returned to $\mathcal{A}_1$, we can intertwine the queries to and responses from the PUFs in the log sequence to each local step of P_2 (in the correct order) and simply move them to the correct position in the view for P_1', replacing the special transparency messages.
- We can proceed analogously for the adversary's final `retro-acc` queries and move the queries accordingly.

Denote this uniquely determined re-ordering process of entries in the view by Unfold and the output when applied to a concrete view by $\mathsf{Unfold}(\mathtt{view})$.

To argue security of our simulator S_1' we denote by $\mathcal{A}_1^{\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]}(x,y)$ the random variable describing the view of our partially malicious adversary $\mathcal{A}_1$ in protocol $\Pi^{\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]}$ for input (x,y), and by $\mathcal{A}_1^{\mathsf{SW}}(x,y)$ the view if we replace all PUF evaluations by software simulations. Similarly, let $S_1^{\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]}(x,f_1(x,y))$ and $S_1^{\mathsf{SW}}(x,f_1(x,y))$ denote the views generated by S_1 for the corresponding oracle. By simulatability and the security of the simulator S_1 for $\Pi^{\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]}$ we have that $\mathcal{A}_1^{\mathsf{SW}}$ and S_1^{SW} are computationally indistinguishable, by using the following hops:[7]

$$\mathcal{A}_1^{\mathsf{SW}}(x,y) \overset{c}{\approx} \mathcal{A}_1^{\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]}(x,y) \overset{c}{\approx} S_1^{\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]}(x,f_1(x,y)) \overset{c}{\approx} S_1^{\mathsf{SW}}(x,f_1(x,y)).$$

Note that $\mathsf{Unfold}(\mathcal{A}_1^{\mathsf{SW}}(x,y))$ has the same distribution as the view of P_1' in Π' for protocol input (x,y); and that $\mathsf{Unfold}(S_1^{\mathsf{SW}}(x,f_1(x,y)))$ has the same distribution as our simulator S_1' for input $(x,f_1(x,y))$. Using once more an informal notation we thus obtain:

$$\mathtt{view}_1^{\Pi'}(x,y) \overset{s}{\equiv} \mathsf{Unfold}(\mathcal{A}_1^{\mathsf{SW}}(x,y)) \overset{c}{\approx} \mathsf{Unfold}(S_1^{\mathsf{SW}}(x,f_1(x,y))) \overset{s}{\equiv} S_1'(x,f_1(x,y)).$$

With the same approach and argument we can now build a secure simulator S_2' for P_2' in Π' from the given simulator $S_2^{\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]}$ for P_2 in $\Pi^{\mathsf{retro}\text{-}\mathcal{M}[\mathcal{P}]}$. □

Note that the above argument applies equally well, even if the adversary does not have retrospective access, if all PUFs created by a party also end up at the party again when the protocol terminates. Then the last phase of read-outs where retrospective access is used, can be omitted. If the two-party protocol $\Pi^{\mathcal{M}[\mathcal{P}]}$ computes a function f securely w.r.t. partially malicious behavior in the bad PUF model, the PUF is simulatable by SWsim, and each PUF in a protocol execution always returns to its manufacturer eventually, then the protocol Π' in Construction 4 also computes f securely in the semi-honest model.

5 PUF-based OT-Protocols with Security against Stateless Bad/Malicious PUFs and PUFs-inside-PUFs Attacks

Section 5 will now investigate whether OT can be realized securely with *stateless* bad/malicious PUFs in the face of certain realistic attack models. Given the impossibility results for *stateful* malicious PUFs in this paper and elsewhere [8, 29, 4], this question lies at hand; the presented material thus immediately connects to previous sections.

[7] We omit the straightforward formalization.

5.1 The Protocol of Dachman-Soled et al. and a Slight Simplification of It

The starting point for our entire treatment in this Section 5 is a protocol by Dachman-Soled et al. [4] from Crypto 2014, which was designed to realize OT secure against *stateless* malicious PUFs.

The protocol is given for the convenience of readers as Protocol 2 below. It assumes that *all* possibly employed bad/malicious PUFs are stateless, and utilizes two PUFs: One $\mathsf{PUF_S}$ introduced by the Sender, another $\mathsf{PUF_R}$ created by the Receiver. It then *"combines"* these two PUFs into one $\mathsf{PUF}_\oplus$, whose responses $r_i = \mathsf{PUF}_\oplus(c_i)$ on challenges c_i are defined as

$$r_i := \mathsf{PUF}_\oplus(c_i) := \mathsf{PUF_R}(c_i) \oplus \mathsf{PUF_S}(c_i).$$

This shall make the "combined" $\mathsf{PUF}_\oplus$ unclonable and unpredictable both for the Sender and the Receiver. [8] We describe the protocol in a stand-alone setting below, not in its original UC-Setting, which slightly simplifies our presentation.

Protocol 2

(OT-Protocol with Intended Security against Stateless Bad/Malicious PUFs by Dachman-Soled et al. [4]**)**

Preparation Phase:

1. Sender creates $\mathsf{PUF_S}$, and sends $\mathsf{PUF_S}$ to Receiver.
2. Receiver creates $\mathsf{PUF_R}$.
3. Receiver collects a random CRP (c, r) from the *"combined"* $\mathsf{PUF}_\oplus$ (where $r = \mathsf{PUF}_\oplus(c) = \mathsf{PUF_R}(c) \oplus \mathsf{PUF_S}(c)$, see above).
4. Receiver sends $\mathsf{PUF_S}$ and $\mathsf{PUF_R}$ to Sender.

OT-Phase:

The Sender holds two bitstrings s_0, s_1. The Receiver holds a choice bit b.

1. Sender randomly chooses a pair of strings $x_0, x_1 \in \{0,1\}^\lambda$, and sends x_0, x_1 to Receiver.
2. Receiver defines $v := c \oplus x_b$, and sends v to Sender.
3. Sender defines $c_0 := v \oplus x_0$ and $c_1 := v \oplus x_1$.
   He applies c_0 and c_1 to $\mathsf{PUF}_\oplus$, collecting responses r_0, r_1.
4. Sender defines $S_0 := s_0 \oplus r_0$ and $S_1 := s_1 \oplus r_1$, and sends S_0, S_1 to Receiver.
5. Receiver outputs $s_b := S_b \oplus r$.

Simplification of the Protocol. Before we discuss the security of the protocol later on, there is one observation that we want to share. Interestingly, the unclonability and unpredictability of the "combined" $\mathsf{PUF}_\oplus$ against a cheating Sender is not exploited in the above Protocol 2 of Dachman-Soled et al. [4]: After the preparation phase, the Sender holds both $\mathsf{PUF_S}$ and $\mathsf{PUF_R}$ (i.e., $\mathsf{PUF}_\oplus$) until protocol completion. It is therefore irrelevant if $\mathsf{PUF}_\oplus$ is unclonable or unpredictable for the Sender, since he has physical access to it until the end of the protocol anyway. In turn, it is irrelevant if $\mathsf{PUF_R}$, which is a "part" of the

[8] We remark that the idea to combine a possibly malicious PUF with a benign PUF via an XOR operation in order to obtain a benign PUF was first used in a PUF-based bit commitment protocol by van Dijk and Rührmair from 2012 [8], which we already mentioned earlier (and which Dachman-Soled et al. [4] unfortunately missed).

"combined" $\mathsf{PUF}_\oplus$, is unpredictable or unclonable for the Sender. Once more in turn, it becomes irrelevant if PUF_R is being used in the protocol at all.

This observation allows the following simplification of Dachman-Soled at al. It uses only one PUF_S instead of two PUFs, saving one PUF (namely PUF_R), and saving one PUF-transfer (namely the transfer of PUF_R from the Receiver to the Sender).

Protocol 3

(Simplified Version of the OT-Protocol of Dachmal-Soled et al. [4], using one PUF instead of two)

Preparation Phase:

1. Sender creates PUF_S, and sends PUF_S to Receiver.
2. Receiver collects a random CRP (c, r) from PUF_S.
3. Receiver returns PUF_S to Sender.

OT-Phase:

The Sender holds two bitstrings s_0, s_1. The Receiver holds a choice bit b.

1. Sender randomly chooses a pair of strings $x_0, x_1 \in \{0,1\}^\lambda$, and sends x_0, x_1 to Receiver.
2. Receiver defines $v := c \oplus x_b$, and sends v to Sender.
3. Sender defines $c_0 := v \oplus x_0$ and $c_1 := v \oplus x_1$.
   He applies c_0 and c_1 to PUF_S, collecting responses r_0, r_1.
4. Sender defines $S_0 := s_0 \oplus r_0$ and $S_1 := s_1 \oplus r_1$, and sends S_0, S_1 to Receiver.
5. Receiver outputs $s_b := S_b \oplus r$.

Brief Security Discussion of Protocols 2 and 3 and Importance of PUF Authentication. Let us lead a short security discussion for Protocols 2 and 3. First of all, it is not too difficult to see that both protocols have equivalent security properties, i.e., the same security guarantees, but also the same attack surface. A formal proof of this fact is deferred to the full version.

Both Protocols 2 and 3 must assume that all used bad PUFs are stateless. Otherwise, a cheating Sender could build a challenge-logger into PUF_S, which he could read out when PUF_S returns to him in the preparation phase. This would allow him to learn the challenge c applied by the Sender. Knowing c, he can deduce the choice bit b from the value v. We remark here as a caveat that we know of no effective way to enforce in practice that bad PUFs are stateless. Even Pappu et al.'s optical PUF [31] can be manipulated to be stateful by spraying a light-sensitive layer on top of it, which changes locally when being illuminated with a laser beam [40]. This mechanism can serve as a covert challenge-logger. Things are even worse for integrated PUFs, which only communicate with the users via a digital challenge-response interface: What is behind the interface is impossible to inspect of verify with the capabilities of average users. Even if PUFs would be opened and inspected invasively, this would destroy them, and would not allow them to be used further [40].

Another important observation is that in both Protocols 2 and 3, the Receiver must not be able to exchange PUF_S, which he receives in the prepration phase, unnoted against a bad $\mathsf{PUF}^*_\mathsf{S}$. If he could, then he might replace the two benign PUF_R and PUF_S by two bad PUFs $\mathsf{PUF}^*_\mathsf{R}$ and $\mathsf{PUF}^*_\mathsf{S}$, which are both simulatable to him. This would allow the Receiver to simulate the "combined" $\mathsf{PUF}_\oplus$, i.e., to simulate both responses r_0 and r_1, to decode both S_0 and S_1, and to learn both secrets s_0 and s_1.

Dachman-Soled et al. [4] make this observation, too. They argue that effective authentication of the $\mathsf{PUF_S}$ is necessary when it returns to the Sender, and suggested that this could be done by the Sender probing a random point (=CRP) of the $\mathsf{PUF_S}$ and then checking it again later [4].

However, it turns out over the next section that CRP-based authentication of the returning $\mathsf{PUF_S}$ is more intricate than it seems at first glance.

5.2 PUF Modifications and PUFs-inside-PUFs Attacks

We start our analysis by suggesting and motivating two new attack models, namely *"PUF modifications"* and *"PUFs-inside-PUFs attacks"*. Recall that in the course of a PUF-protocol, PUFs may be physically transfered multiple times between users; the above Protocols 2 and 3 can serve as examples for this. Many current attack models [4] assume that adversaries might attempt to use malicious PUFs instead of benign PUFs from the start, or to completely exchange benign PUFs against malicious PUFs during the protocol run.

However, adversaries could also *partially modify* existing PUFs (in *"PUF modification attacks"*), or might even use existing PUFs *physically within* the new bad PUF constructs (in *"PUFs-inside-PUFs attacks"*).

Concrete examples include:

- The adversary might try to add certain physical parts, or also certain code in the case of FPGA-PUFs, to a given PUF which *"logs"* or *"records"* every challenge (and possibly every response), turning a benign PUF into a challenge-logging PUF. Note that the challenge-response behavior of the original PUF thereby is not altered.
- He might add new hardware parts (or code in the case of FPGA-PUFs) that communicate the applied challenges (and/or the responses) to the adversary, turning a benign PUF into a communicating PUF. Again, the modification leaves the CRP-behavior of the original PUF unchanged.
- In a similar manner, he might try to create a modified, stateful and communicating PUF, that (for example) *adapts* its CRP-behavior after communication with the adversary.
- The adversary might even use existing, "old" PUFs as subparts of a new, malicious PUF. Such *"PUFs-inside-PUFs attacks"* are a subclass of PUF modifications, and represent a particularly dangerous attack form, as we will see below. One example of a simple PUF-inside-PUF construction is illustrated in Figure 1.

The applicability of PUF modification and PUFs-inside-PUFs attacks obviously depends on the exact usage scenario, the employed type of PUF, and the capabilities of the adversary. But if the used PUF is *"integrated"*, i.e., if the honest users communicate with the PUF merely over a digital CRP-interface, the majority of the attacks are indeed hard to prevent. Recall that what is behind the CRP-interface is difficult to check or inspect for honest users without opening and thus destroying the PUF [40]. As integrated PUFs represent the vast majority of current PUF architectures, this makes the PUF modification and PUFs-inside-PUFs attacks particularly pressing.

Non-integrated PUFs, such as non-integrated optical PUFs a la Pappu et al. [31], are only partly concerned by the above type of attacks. On the one hand, some above attacks are indeed less realistic; for example, so-called "communicating bad PUFs" [40] seem hard to realize in connection with non-integrated optical PUFs. On the other hand, also non-integrated optical PUFs can be modified maliciously in certain ways [40]. For example, they

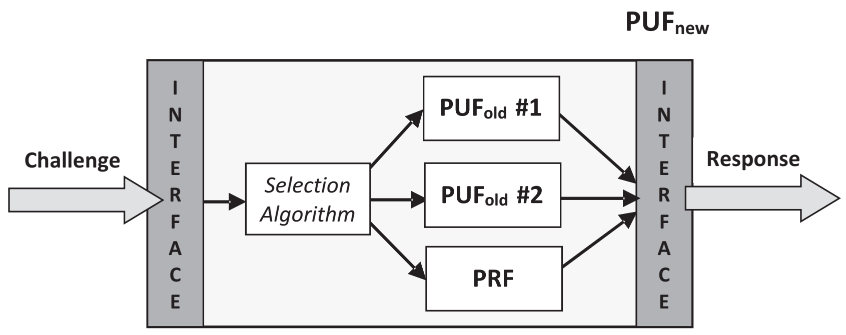

Fig. 1. A PUFs-inside-PUFs example construction: Given two existing, "old" PUFs, the adversary constructs a "new", bad/malicious PUFs, which contains the old PUFs as subparts. In this example, the new bad PUF receives a challenge over its digital input interface. It internally passes on the challenge to a selection algorithm, which decides to which of its subcomponents this challenge shall be applied: To one of the old PUFs, or to a digital pseudo-random function that is known to the adversary. The output of the selected component is then passed on to the outside via the digital response interface of the new PUF.

could be made stateful by spraying a light-sensitive layer on top of them. The layer could change locally when it is hit by a laser beam, "marking" the point of incidence of the optical PUF's challenge, and realizing some form of challenge-logging behavior [40].

Several further arguments support our PUFs-inside-PUFs attack model. Firstly, already the PUF attack models of Dachman-Soled et al. [4] themselves assume that newly created PUFs can be used as subparts of bad PUFs. Why should old, existing PUFs not be used in the same way? Secondly, the use of *"tokens-inside-tokens"* is already established in the classical hardware token literature [15]. Finally, also communicating tokens have been investigated in the classical token literature recently [9]. All of these reasons make the abovementioned PUF modifications, in particular PUFs-inside-PUFs, a legitimate scenario.

5.3 Problems of CRP-based PUF-Authentication under PUFs-inside-PUFs Attacks

In order to prevent PUF-modification, PUFs-inside-PUFs attacks, and PUF substitution, it seems suggestive to authenticate PUFs whenever they return home in the course of a PUF-protocol. To this end, one might apply a few precorded challenges in sequence (i.e., in a consecutive block) to the PUF, and check if the obtained responses match the prerecorded values. The same technique has also been suggested, for example, by Dachman-Soled et al. [4]. We observe in this section, however, that this approach is dangerous. Both stateful and stateless PUFs-inside-PUFs can be constructed that survive certain CRP-based authentication. Our stateless PUFs-inside-PUFs constructions lead to an attack vector on the OT-protocol by Dachman-Soled et al. [4] in the PUFs-inside-PUFs model, which will be described in Section 5.3.3.

5.3.1 Stateful PUFs-inside-PUFs that Survive Blockwise CRP-Based Authentication with Non-Negligible Probability. Let us start by the stateful case. We assume

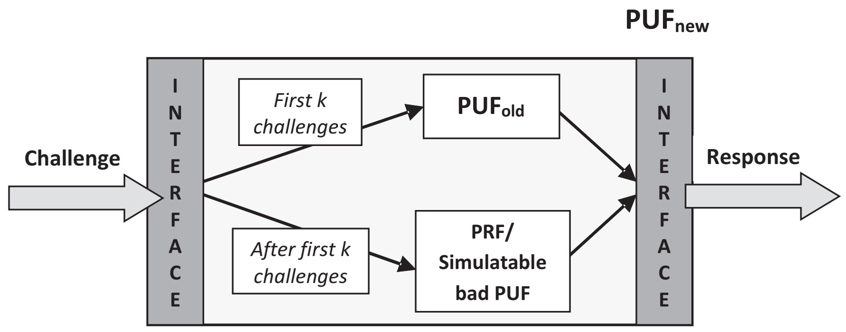

Fig. 2. An example of a stateful PUFs-inside-PUFs construction, which internally employs an old PUF. It cannot be distinguished from the old PUF by applying k CRPs in a consecutive block at the beginning of a protocol. Still, it potentially allows cheating, because from the $k+1$-th challenge onwards, it outputs only responses that are simulatable to the adversary.

that the adversary holds some existing $\mathsf{PUF}_{\mathsf{old}}$, and would like to construct a stateful, malicious $\mathsf{PUF}_{\mathsf{new}}$, which has the following properties:

- When $\mathsf{PUF}_{\mathsf{new}}$ is returned to the honest users/the creator of $\mathsf{PUF}_{\mathsf{old}}$, it will pass a standard authentication test, in which a block of k prerecorded challenges is applied to $\mathsf{PUF}_{\mathsf{new}}$, and its responses are compared to some prerecorded responses of $\mathsf{PUF}_{\mathsf{old}}$. I.e., honest users will not notice the difference between $\mathsf{PUF}_{\mathsf{new}}$ and $\mathsf{PUF}_{\mathsf{old}}$ when applying a standard, blockwise, CRP-based authentication step.
- After the authentication phase with the k prerecorded CRPs has been passed, $\mathsf{PUF}_{\mathsf{new}}$ will only output responses that are simulatable/predictable to the adversary (allowing him to cheat in protocols).

The described behavior can be achieved very easily in the PUFs-inside-PUFs model. Figure 2 gives a simple construction of a stateful $\mathsf{PUF}_{\mathsf{new}}$ from an existing $\mathsf{PUF}_{\mathsf{old}}$ which has the desired properties. $\mathsf{PUF}_{\mathsf{new}}$ passes the abovementioned blockwise authentication test with probability 1 if the parameter k is known to the adversary. Even in the case that the parameter k is unknown a priori to the adversary, the correct value k may be guessed correctly with non-negligible probability by him in the preparation of $\mathsf{PUF}_{\mathsf{new}}$. Alternatively, it may even be guessed by the bad PUF $\mathsf{PUF}_{\mathsf{new}}$ itself during the protocol, assuming that $\mathsf{PUF}_{\mathsf{new}}$ has some sort of randomness on board. We comment that only polynomial values for the number k of prerecorded CRPs in the authentication are admissible, since otherwise the preparation and the authentication phase would both have superpolynomial runtime. Guessing the value for k correctly therefore can be done with non-negligible probability.

5.3.2 Stateless PUFs-inside-PUFs that Survive Blockwise CRP-Based Authentication with Non-Neglibible Probability. The above problems more or less persist in the stateless case. The mere assumption that all bad PUF are stateless, which was introduced in [4], hence cannot restore the possibility for secure CRP-based authentication.

Let us sketch our stateless PUFs-inside-PUFs construct. Again, the goal of the adversary is to build some $\mathsf{PUF}_{\mathsf{new}}$ which contains a given $\mathsf{PUF}_{\mathsf{old}}$, following the PUFs-inside-PUFs

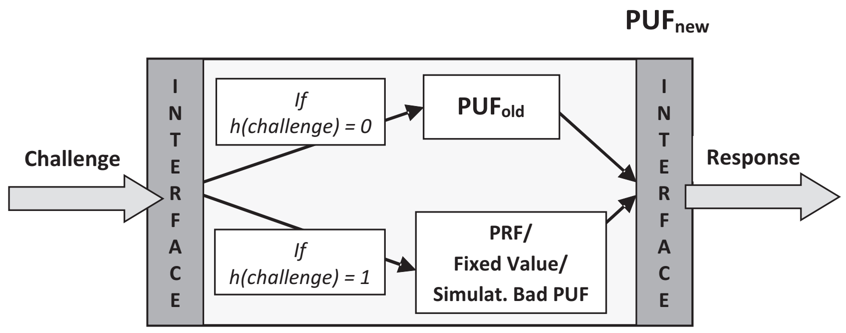

Fig. 3. Construction of a stateless Bad PUF that follows the "PUFs-inside-PUFs" paradigm. It will survive an authentication phase in which k authenticating challenges are applied in sequence and are tested for correctness afterwards, and will subsequently output a simluatable response value known to the adversary, with non-negligible probability.

attack model. This $\mathsf{PUF}_{\mathsf{new}}$ shall go unnoticed in a CRP-based authentication step, in which k challenges are applied in one block to $\mathsf{PUF}_{\mathsf{new}}$.

In the stateful case, $\mathsf{PUF}_{\mathsf{new}}$ was able to count the challenges that were applied, and could switch its behavior after k challenges. In opposition to this, stateless PUFs cannot count. However, counting can to some extent be emulated by using a binary, $(k+1)$-wise independent hash function h with the following property:

1. With probability $\frac{k}{k+1}$, the output of $h(x)$ is equal to zero.
2. With probability $1 - \frac{k}{k+1} = \frac{1}{k+1}$, the output of $h(x)$ is equal to one.

The $\mathsf{PUF}_{\mathsf{new}}$, which contains $\mathsf{PUF}_{\mathsf{old}}$, then can be programmed to have the following functionality:

1. Given a challenge c as input, it computes $h(c)$, using the above hash function h.
2. If $h(c) = 0$, it re-directs the challenge to $\mathsf{PUF}_{\mathsf{old}}$, and outputs the response it receives from $\mathsf{PUF}_{\mathsf{new}}$.
3. If $h(c) = 1$, it outputs a fixed value known to its creator and to the adversary, for example the all zero string $0 \cdots 00$. (Alternatively, it may apply a pseudorandom function PRF known to the adversary to the input challenge c, and output $\mathsf{PRF}(c)$, or forward the challenge to any other type of simulatable bad PUF, outputting the response of the latter.)

The resulting bad PUF is illustrated in Figure 3. It survives blockwise authentication with a probability that is non-negligible in k. Once more, if k is not known to the adversary at the time when he creates $\mathsf{PUF}_{\mathsf{new}}$, he can simply guess it, being correct with non-negligible probability. This leads to a non-negligible probability that $\mathsf{PUF}_{\mathsf{new}}$ will survive the authentication phase, and will output a manipulated response value right after.

A more detailed analysis follows. Consider the following $(k, \mathcal{D})$-*simple consistency game* for $\mathcal{M}[\mathcal{P}]$. The game measures the adversary's success probability to be able to pass k consistency/authentication checks with a maliciously created PUF in sequence, and to "program" the next PUF-output on a certain value, here demanding for concreteness that this output shall simply be an all zero string 0^λ. If the adversary succeeds in this (with non-negligible probability), this usually suffices to break any cryptographic scenario in which this sub-protocol is used:

- The challenger creates a PUF via `create`(pid) and computes k responses

$$r_i = \texttt{evaluate}(pid, c_i)$$

  for random challenges $c_i \leftarrow \mathcal{D}(1^\lambda)$, sampled independently according to distribution $\mathcal{D}$.
- The challenger issues a `handover`(pid) and the efficient adversary may now call

$$\texttt{set-algo}(\mathrm{pid}, A),$$

  where A is an arbitrary stateless oracle circuit with oracle access to $\mathcal{P}$. In particular, A may call the original PUF with identifier pid. The adversary calls `handover`(pid) again.
- The challenger samples $c \leftarrow \mathcal{D}(1^\lambda)$ and the adversary wins if we have $\texttt{evaluate}(\mathrm{pid}, c_i) = r_i$ for all $i = 1, 2, \ldots, k$, but if next $\texttt{evaluate}(\mathrm{pid}, c) = 0^\lambda$.

We show that for any polynomially bounded k and for any distribution $\mathcal{D}$ with super-logarithmic min-entropy there is an adversary winning the consistency game with probability $\Omega(1/k)$. Our adversary $\mathcal{A}$, when receiving the PUF with identifier pid, creates algorithm A as follows: A includes a $(k+1)$-wise independent binary hash function h mapping to 0 with probability $1 - \frac{1}{k+1}$, and to 1 with probability $\frac{1}{k+1}$. Algorithm A on input c first computes $h(c)$. If $h(c) = 0$ then A returns the original value $\texttt{PUFeval}(\mathrm{pid}, c)$; else, for $h(c) = 1$, the algorithm returns 0^λ.

To analyze $\mathcal{A}$'s success probability note that for each of the k test queries of the challenger, the probability that the malicious PUF returns the original value, is $1 - \frac{1}{k}$. Furthermore, for distinct challenges the outputs of the $(k+1)$-wise independent hash functions are independent. Since the challenger makes at most k different test queries, possibly repeating some queries by chance, the probability of answering all of the distinct queries consistently (with the original PUF) is at least $(1 - \frac{1}{k+1})^k \geq \Omega(e^{-1})$. Note further that the probability of c being different from all $c_1, \ldots, c_k$ is at least $\frac{1}{2}$ by the super-logarithmic min-entropy of $\mathcal{D}$. Hence, with constant probability the final evaluation happens on a fresh value c and such that all previous test queries used the original PUF. Given this, by the $(k+1)$-wise independence of h, we obtain the output 0^λ for c with probability $\frac{1}{k+1}$, making the overall success probability at least $\Omega(1/k)$.

We comment that the above game assumes that k is known to the adversary. Still, our strategy even works if k was chosen randomly during the protocol run by the honest parties. To see this, note that we may assume that k is chosen from the range $\{1, \ldots, N\}$, where N is polynomial in the security parameter λ. [9] The adversary can hence guess a value k', hoping that it is equal to the later choice of k. His chances for doing so are polynomial in λ. This means that the above attack still has a non-negligible chance of success.

5.3.3 Security Analysis of the OT-Protocol by Dachman-Soled et al. under Stateless PUFs-inside-PUFs Attacks.

The existence of PUFs-inside-PUFs attacks has consequences for the security of the OT-protocol of Dachman-Soled et al. [4] (see Protocol

[9] A choice of k that is superpolynomial in λ is not admissible, since the protocol then would have superpolyomial runtime.

2 above): It turns out that it can be attacked by the stateless bad PUFs-inside-PUFs of Section 5.3.2 and Figure 3. Before proceeding with the details, we would like to stress once more that PUFs-inside-PUFs lie **outside of** the original attack model of Dachman-Soled et al. [4].

Let us now detail said attack. First of all, recall from Section 5.1, page 23 that the protocol of Dachman-Soled et al. [4] is not secure if a cheating Receiver can replace the PUF_S (which he receives from the Sender in the preparation phase) by a bad $\mathsf{PUF}^*_\mathsf{S}$ which is simulatable to the Receiver. The reason is that under this assumption, and under the further hypothesis that the Receiver will not use an honest PUF_R as the second PUF in the protocol, but again a bad $\mathsf{PUF}^*_\mathsf{R}$ that is simulatable to him, the entire "combined" $\mathsf{PUF}_\oplus$ becomes simulatable for the Receiver. This allows him to cheat.

This necessitates authentication of PUF_S when it returns to the Sender, as already observed by Dachman-Soled et al. They suggest that (quote) *"this can be done by having the Sender probe a random point before sending* PUF_S *and then checking it again later. We omit this check from the [protocol]"* [4]. In other words, they stipulate that PUF_S shall be authenticated by applying a prerecorded challenge to it, and by comparing the given responses to a prerecorded responses. Slightly generalizing one could use a polynomial number of k prerecorded CRPs, and test them consecutively in a block of k challenges (Dachman-Soled et al.'s original idea being the special case $k = 1$).

It is not too difficult to imagine how Dachman-Soled et al.'s protocol is going look like when such an authentication step is included; to be fully specific, and to allow an accurate description of our attack, we will nevertheless put it down below.

Protocol 4

(OT-Protocol of Dachman-Soled et al. [4] with Additional CRP-based Authentication Step)

Preparation Phase:

1. Sender creates PUF_S.
2. Sender collects k random *"authenticating"* CRPs $(c^*_1, r^*_1), \ldots, (c^*_k, r^*_k)$ from PUF_S.
3. Sender sends PUF_S to Receiver.
4. Receiver creates PUF_R.
5. Receiver collects a random CRP (c, r) from the *"combined"* $\mathsf{PUF}_\oplus$ (where $r = \mathsf{PUF}_\oplus(c) = \mathsf{PUF}_\mathsf{S}(c) \oplus \mathsf{PUF}_\mathsf{R}(c)$, as before).
6. Receiver sends PUF_S and PUF_R to Sender.

OT-Phase:

The Sender holds two bitstrings s_0, s_1. The Receiver holds a choice bit b.

1. *Authentication Step:*
   - Sender applies the k challenges $c^*_1, \ldots, c^*_k$ to PUF_S.
   - He compares the obtained responses to the responses $r^*_1, \ldots, r^*_k$ prerecorded in Step 2 of the preparation phase.
   - If they match, he continues. Else, he aborts.
2. Sender randomly chooses a pair of strings $x_0, x_1 \in \{0,1\}^\lambda$, and sends x_0, x_1 to Receiver.
3. Receiver defines $v := c \oplus x_b$, and sends v to Sender.

4. Sender defines $c_0 := v \oplus x_0$ and $c_1 := v \oplus x_1$.
   He applies c_0 and c_1 to $\mathsf{PUF}_\oplus$, collecting responses r_0, r_1.
5. Sender defines $S_0 := s_0 \oplus r_0$ and $S_1 := s_1 \oplus r_1$, and sends S_0, S_1 to Receiver.
6. Receiver outputs $s_b := S_b \oplus r$.

We note again that Dachman-Soled et al. originally suggest using one CRP in the authentication phase, i.e., they deal with the case $k = 1$. However, regardless of the value of k, the above PUF-authentication mechanism of k blockwise applied CRPs can be outsmarted by PUFs-inside-PUFs as follows:

– Instead of honestly employing a benign PUF_R, the malicious Receiver from the start of the protocol uses a bad PUF $\mathsf{PUF}^*_\mathsf{R}$ which is simulatable to him, e.g., which contains a pseudo-random function PRF_1.
– When the Receiver is handed PUF_S from the Sender, he tries to guess the value of k (i.e., the number of CRPs used for authentication). Since k is polynomial, he has a non-negligible chance of success. Subsequently, he constructs a new, bad $\mathsf{PUF}^*_\mathsf{S}$ following Figure 3, using PUF_S as the "old" PUF.
– More specifically, $\mathsf{PUF}^*_\mathsf{S}$ shall apply a $(k+2)$-wise independent hash function h with a 1-bit output in order to decide whether any applied challenge c shall be passed on to PUF_S or to some pseudo-random function PRF_2. When $h(c)$ is equal to 0, the challenge shall be passed on to PUF_S. When $h(c)$ is equal to 0, the challenge shall be passed on to PRF_2. h shall furthermore have the property that it outputs 0 with probability $\frac{k}{k+1}$ and 1 with probability $\frac{1}{k+1}$.
– The malicious Receiver then hopes that the following two events will happen in the course of the protocol: First of all, the internal hash function h of $\mathsf{PUF}^*_\mathsf{S}$ will take the value 0 for all of the k challenges $c^*_1, \ldots, c^*_k$ that are applied to $\mathsf{PUF}^*_\mathsf{S}$ by the Sender in the blockwise authentication step (compare Protocol 4 above). Secondly, h will take the value 1 for the two challenges c_0 and c_1 that are applied to $\mathsf{PUF}^*_\mathsf{S}$ in the further course of the protocol.
– As our discussion in Section 5.3.2 shows, these two events happen jointly with non-negligible probability in k. If the two events happen, the Receiver can cheat, though: First of all, his substitution of $\mathsf{PUF}^*_\mathsf{S}$ for PUF_S will go unnoticed. Secondly, he can simulate the responses r_0 and r_1 of $\mathsf{PUF}_\oplus$ (which are equal to $\mathsf{PRF}_1(\mathsf{c}_0) \oplus \mathsf{PRF}_2(\mathsf{c}_0)$ and $\mathsf{PRF}_1(\mathsf{c}_1) \oplus \mathsf{PRF}_2(\mathsf{c}_1)$). This allows him to decode both S_0 and S_1 and to learn both secrets s_0 and s_1.

This completes the exposition of our attack.

5.4 The Countermeasure of Interleaved Challenges, and a First Protocol Attempt (Which Fails)

Given the issues of PUF-authentication and the vulnerability of Dachman-Soled et al. [4], it is natural to ask for countermeasures. How can effective authentication of PUFs be achieved in the face of PUFs-inside-PUFs attacks?

One possibility consists of *"randomly interleaved challenges"*. Let us assume that some given PUF-protocol applies m challenges to the PUF during its normal protocol run (let us call these challenges *"protocol challenges"*). Suppose further that we would like to add an authentication step to the protocol, which shall detect PUFs-inside-PUFs constructs. The authentication step shall consist of the application of k *"authenticating challenges"* and of checking the responses for correctness.

Now, the basic idea of *"interleaved challenges"* is as follows: Instead of executing the authentication step *in a consecutive block* somewhere during the protocol, in which the k "authenticating"-challenges in are applied in strict sequence to the PUF, we instead *mix* or *interleave* the protocol-challenges and the authenticating-challenges. Their exact ordering is chosen randomly during the protocol run.

The adversary then only has a chance of $1/\binom{m+k}{k}$ to correctly guess the k positions in which authenticating challenges will be applied and the m positions in which the protocol challenges will be applied. If m and k are about the same size, this chance will be exponentially small. This is in stark contrast to situation where a *single* authenticating challenge or a *fixed block* of authenticating challenges is applied: There, the adversary has a non-negligible probability of guessing the starting and ending point of the block correctly, and to thus guess the nature of all applied challenges.

Let us conclude by two comments. First, we stress again that it is essential for our technique of interleaved challenges that the number of protocol challenges m and the number of authenticating challenges k are on the same order. Only then, the number of interleaving arrangements $\binom{m+k}{k} = \binom{m+k}{m}$ becomes exponential. Secondly, when m and k are on the same order, interleaved challenges are an authentication method that prevents not only the specific *stateless* PUFs-inside-PUFs of Figure 3, but also the specific *stateful* PUFs-inside-PUFs of Figure 2. [10]

5.4.1 A First Protocol Attempt, Which Fails. The above discussion suggests to combine of the idea of interleaved challenges with the original OT-protocol of Dachman-Soled et al. [4] in order to obtain security against PUFs-inside-PUFs attacks in the stateless PUF scenario. The resulting protocol is given for illustration purposes as Protocol 6 in Appendix A.

Unfortunately, there is a very subtle attack on this protocol, which we also provide in Appendix A. In a nutshell, Protocol 6 can be attacked since a cheating Receiver has some chance to maliciously influence the challenges that are applied to the PUF by the Sender. He can enforce that the challenges have a certain format, which will trigger a certain type of misbehavior in the PUF. A similar observation has already been made by Rührmair and van Dijk in their quadratic attacks [38, 39] on the OT-protocol by Brzuska et al. [1]. Full details can be found in Appendix A.

This leaves the question whether PUF OT-protocols that remain secure in the face of malicious, stateless PUFs and PUFs-inside-PUFs attacks can be devised.

5.5 An OT-Protocol for Stateless Bad PUFs that remains Secure under PUFs-inside-PUFs Attacks

This section will finally present a protocol secure against stateless bad/malicious PUFs, including PUFs-inside-PUFs attacks. The protocol combines interleaved challenges with interactive hashing (IH) [27]. Furthermore, it makes use of our simplifying observation from

[10] This does not mean that all protcols could be secured against stateful bad PUFs by interleaving challenges, though: Recall that there are other PUF modifications where "good" PUFs are turned into stateful bad PUFs that do not change the challenge-response behavior of the original PUF at all, and hence cannot be detected by CRP-based PUF-authentication. One straightforward example is the malicious addition of a challenge logger to an existing PUF.

Section 5.1/Protocol 3, and uses only one PUF. As an interesting historic sidenote, also the first PUF-based OT-protocol from 2010 employed IH [34], and so did the PUF-based OT-schemes from CHES 2012 [38] and JCEN 2013 [39].

In a nutshell, IH is a two-party protocol where the first player has no input, and the second player holds an m-bit string C. As output, both players obtain two m-bit strings C_1 and C_2, with the property that one of these strings is equal to C, and the other one is essentially random. Even if the players cheat, the first player cannot learn which of the strings C_1, C_2 is equal to C, and the first player cannot influence which value the "other" string unequal to C will take; it is guaranteed by the protocol to be random. For further details, we refer to the standard literature on IH [27, 44].

Our protocol runs as follows.

Protocol 5

(OT-Protocol with Security against Stateless Bad PUFs, including PUFs-inside-PUFs Attacks)

Preparation Phase:

1. Sender creates $\mathsf{PUF_S}$.
2. Sender collects k random *"authenticating"* CRPs $(c_1^*, r_1^*), \ldots, (c_k^*, r_k^*)$ from $\mathsf{PUF_S}$.
3. Sender sends $\mathsf{PUF_S}$ to Receiver.
4. Receiver collects k random *"protocol"* CRPs $(c_1, r_1), \ldots, (c_k, r_k)$ from $\mathsf{PUF_S}$.
5. Receiver returns $\mathsf{PUF_S}$ to Sender.

OT-Phase:

The Sender holds two bitstrings s_0, s_1. The Receiver holds a choice bit b.

1. *Interactive Hashing Step:*
   - Sender and Receiver get engaged in an interactive hashing (IH) protocol.
   - Receiver's (secret) input to the IH-protocol is $C = (c_1, \ldots, c_k)$.
   - The IH-output of both parties are two k-tuples of PUF-challenges $C(0) = (c_1^0, \ldots, c_k^0)$ and $C(1) = (c_1^1, \ldots, c_k^1)$.
   - By the properties of IH, exactly one of the two tuples $C(0), C(1)$ is equal to C. Let $I \in \{0, 1\}$ be the index for which $C(I) = C$.
2. Receiver sends $b' := b \oplus I$ to Sender.
3. *Challenge Mixing Step:*
   - Sender at this point knows the following $3k$ challenges:
   $$c_1^*, \ldots, c_k^*, c_1^0, \ldots, c_k^0, c_1^1, \ldots, c_k^1.$$
   - He applies these challenges in a **randomly permuted order** to $\mathsf{PUF_S}$, collecting (in randomly permuted order) the corresponding responses
   $$r_1^*, \ldots, r_k^*, r_1^0, \ldots, r_k^0, r_1^1, \ldots, r_k^1.$$
4. *Authentication Step:*
   - Sender compares the responses $r_1^*, \ldots, r_k^*$ obtained in the last step to the responses of the prerecorded *"authenticating"* CRPs of Step 2 of the preparation phase.

  - If they match, he continues. Else, he aborts.

5. *Blinding of OT-strings with k responses:* Sender computes

$$S_0 := s_0 \oplus r_1^{b'} \oplus \ldots \oplus r_k^{b'},$$

and

$$S_1 := s_1 \oplus r_1^{1-b'} \oplus \ldots \oplus r_k^{1-b'},$$

and sends S_0, S_1 to the Receiver.

6. Receiver produces as output $s_b := S_b \oplus r_1 \oplus \ldots \oplus r_k$.

Brief Security Discussion. We suggested earlier that healthy PUF protocol design should explicitly name the security assumptions and possible attack surfaces of the protocol. So we briefly do in this section. Users of the protocol must carefully decide whether all of the following conditions are met in their specific application scenario:

- The Sender must not learn *any single* of the challenges c_i that are applied by the Receiver in the preparation phase. This implies, first of all, that $\mathsf{PUF_S}$ has no challenge-logger [8, 40] that the Sender could read out after the PUF has returned to him. (This is enforced in theory by our assumption that $\mathsf{PUF_S}$ is stateless. But it must also be guaranteed in practice by users, which is less trivial: How would the Receiver detect the existence of a challenge logger when he receives $\mathsf{PUF_S}$?).
  Furthermore, even if $\mathsf{PUF_S}$ is stateless, there must be no direct communication between $\mathsf{PUF_S}$ and the Sender (or the PUF-embedding hardware and the Sender), by which the Sender would learn one or more of the c_i. In other words, $\mathsf{PUF_S}$ must not be a so-called *"communicating PUF"*. Note that $\mathsf{PUF_S}$ does not need to be stateful for such communication. If the Sender only learns one single of the challenges c_i, this suffices to derive the index bit I, and to learn the Receiver's choice bit.
- The Receiver must not learn *all* of the responses r_i^0 and r_i^1. Since he already knows all r_i^I, this means that he must not learn all r_i^{1-I}. This obviously implies that $\mathsf{PUF_S}$ must be a secure Strong PUF, e.g., that it is resilient against modeling attacks, that its large number of CRPs prevents a full read-out of the entire CRP-space, and that it cannot be physically cloned. It further necessitates that the Receiver must not be able to turn $\mathsf{PUF_S}$ into a so-called *"communicating"* $\mathsf{PUF_S^*}$ [8, 40] before he returns it to the Sender, which communicates all the r_i^{1-I} to him during or after the protocol. Again, we stress that a communicating PUF may be stateless.

There are a number of other security aspects that should not go unnoticed:

- Similar to all the other Strong PUF protocols of recent years [34, 1, 28, 29, 8, 40, 6], the above protocol only is secure in some type of stand alone setting. In particular, its security can be broken if the Receiver gets access to the used PUF after protocol completion. He will then read out all responses r_i^0 and r_i^1, decode both S_0 and S_1, and learn both secrets s_0 and s_1.
- Interestingly, Protocol 5 will remain secure if the Receiver maliciously turns the stateless $\mathsf{PUF_S}$ into a stateful $\mathsf{PUF_S^*}$ before he returns it to the Sender. For example, the protocol is resilient against a Receiver who uses stateful PUFs-inside-PUFs attacks like the one of Figure 3 – such attempts will be detected by our interleaved challenges.
- Protocol 5 has some *optimality properties* with respect to the following aspects:

(i) *The number of used PUFs:* It uses one PUF, which is obviously minimal, in opposition to Dachman-Soled et al.'s original protocol that employs two PUFs [4].

(ii) *The number of PUF-transfers:* Our protocol has two PUF transfers, which also is minimal. Recall that one single PUF transfer provably cannot create secure OT-Protocols if this PUF is simulatable to its creator [8, 4].

(iii) *The statelessness condition on* $\mathsf{PUF_S}$*:* The $\mathsf{PUF_S}$ must merely be stateless when it is introduced in the protocol, i.e., when the Sender transfers it to the Receiver. It must not be stateless when it returns from the Receiver to the Sender. This means that the Receiver may attempt turning $\mathsf{PUF_S}$ into a stateful PUF-inside-PUF $\mathsf{PUF}^*_\mathsf{S}$, but this still will not allow him to cheat (it will be thwarted by our interleaved challenge method).
Again, this assumption is minimal; if $\mathsf{PUF_S}$ is stateful from the start, containing a challenge-logger for the Sender, the protocol will not be secure (see this paper and [8]).

Finally, we emphasize again that it is essential for the security of Protocol 5 that

- authenticating and protocol challenges are randomly interleaved,
- there is roughly the same number of authenticating challenges $(c^*_1, \ldots, c^*_k)$ and protocol challenges $(c^0_1, \ldots, c^0_k), (c^1_1, \ldots, c^1_k)$, meaning that there are exponentially many possibilities for randomly interleaving both types of challenges,
- and that the protocol has been constructed in such a way that an adversary would have to know *all* the $2k$ protocol responses $(r^0_1, \ldots, r^0_k), (r^1_1, \ldots, r^1_k)$ in order to cheat, i.e., in order to decrypt both S_0 and S_1 and learn both s_0 and s_1. This is achieved by XORing k responses each onto both s_0 and s_1.

A formal proof of the protocol's security based on the above observations is defered to the full version.

6 Conclusion

This paper dealt with the use of so-called Strong PUFs in advanced protocols like oblivious transfer (OT), key exchange (KE), and bit commitment (BC). The corresponding research area has been very active in recent years, with a mixture of new attacks, constructive results, and impossibility theorems being obtained. One reason for the strong activities was that the employment of PUFs, and generally of any hardware tokens, in cryptographic protocols introduces a number of new attack surfaces, some of which had initially been overlooked. These include:

- The adversary's capability to create bad/malicious PUFs from scratch, and to use them instead of the benign PUFs that would be expected by honest users (e.g., [28, 29, 8, 40]);
- the possibility to intercept and substitute PUFs, instead of assuming authentic delivery (e.g., [33, 8, 40, 5]);
- the possibility to partly modify existing PUFs while the adversary has physical access to them, for example building new malicious PUFs by using existing PUFs as subcomponents in so-called *"PUFs-inside-PUFs"* attacks (this paper);
- the possibility that bad/malicious PUFs may communicate with their creators, for example transmitting challenges or responses, or even adapting their challenge-response behavior after communication (see [8, 40, 9]);

– and, finally, the option to retrospectively access PUFs after the completion of a protocol, or even between certain phases of a protocol (for example between the commit and reveal phase in bit commitment) (see [42, 8, 40]).

All of these attack points can be motivated very well by practical and realistic example scenarios. They may require different levels of effort when being implemented by the adversary, depending on the exact usage scenario. But they certainly cannot, and must not, be excluded a priori in PUF protocol design. We advise that comprehensive PUF protocol design should rather make all conceivable attack surfaces and underlying security assumptions explicit, aiming at security under a minimal set of well-achievable assumptions.

We tried to obtain some progress in this situation. We presented a number of impossibility theorems as well as constructive results, and suggested a new attack model termed PUFs-inside-PUFs attacks together with countermeasures against this attack model. Our contributions have been described in all detail in Section 1.4; in a nutshell, they include the following:

– Any PUF that is "ideal" and retrospectively accessible can be replaced by a standard random oracle. By applying the famous Impagliazzo-Rudich result [20], this means that the power of plain Strong PUFs under retrospective access does not suffice to implement KE or OT.
– Any PUF that is both bad/malicious and retrospectively accessible can be completely eliminated from the protocol. The protocol can be compiled into an information-theoretically equivalent one without this PUF.
– We simplify a recent OT-protocol by Dachman-Soled et al. [4] from CRYPTO 2014, which was originally designed to withstand bad PUF use. Our new protocol achieves the same security properties under the same assumptions, but uses only one PUF instead of two.
– We propose the new adversarial model of *"PUFs-inside-PUFs attacks"*, and show that the earlier protocol of Dachman-Soled et al. [4] from CRYPTO 2014 is indeed vulnerable in this model. We stress again that the model is an extension of the original scenario of Dachman-Soled et al. [4], meaning that their proofs remain viable in their original framework.
– We construct a new PUF-based OT-protocol, which is secure against PUFs-inside-PUFs attacks if all used bad PUFs are stateless. Our protocol introduces the technique of interleaved challenges, and uses interactive hashing as a building block. We illustrate why the use of interactive hashing in the protocol is necessary, and why a first protocol attempt that builds on the OT-protocol of Brzuska et al. from CRYPTO 2011 [1], and which does not use interactive hashing, fails.

Our findings also have direct relevance for the PUF hardware community, connecting both worlds. They prove that "plain" Strong PUFs and smart protocol design *alone cannot* establish security if a PUF can be retrospectively accessed. At the same time, such PUF re-use appears ecnomically and practically imperative. This motivates new Strong PUF variants like reconfigurable PUFs [22], erasable PUFs [42, 40], or certificable PUFs. The effective silicon implementation of these new Strong PUF types is mostly open to this date, however. We would like to pose their implementation as a highly rewarding research goal to the PUF hardware community in this work.

Acknowledgments

We would like to thank Marc Fischlin for contributing large parts of Section 5.3.2. Among many other things, he suggested the idea to use k-wise independent hash functions in order to make PUFs-inside-PUFs stateless. Marten van Dijk was supported in part by AFOSR MURI under award number FA9550-14-1-0351.

References

1. Christina Bruzska, Marc Fischlin, Heike Schröder, Stefan Katzenbeisser: *Physically Unclonable Functions in the Universal Composition Framework.* CRYPTO 2011.
2. Ran Canetti: *Universally Composable Security: A New Paradigm for Cryptographic Protocols.* FOCS 2001: 136-145.
3. Ran Canetti, Marc Fischlin: *Universally Composable Commitments.* CRYPTO 2001: 19-40
4. Dana Dachman-Soled, Nils Fleischhacker, Jonathan Katz, Anna Lysyanskaya, Dominique Schröder: Feasibility and Infeasibility of Secure Computation with Malicious PUFs. CRYPTO (2) 2014: 405-420
5. Özgür Dagdelen, Marc Fischlin: Intercepting Tokens in Cryptographic Protocols: The Empire Strikes Back in the Clone Wars ISIT 2014 – IEEE International Symposium on Information Theory, IEEE, 2014.
6. Ivan Damgard, Alessandra Scafuro: Unconditionally Secure and Universally Composable Commitments from Physical Assumptions. ASIACRYPT (2) 2013: 100-119
7. Marten van Dijk: *System and method of reliable forward secret key sharing with physical random functions.* US Patent No. 7,653,197, October 2004.
8. Marten van Dijk, Ulrich Rührmair: Physical Unclonable Functions in Cryptographic Protocols: Security Proofs and Impossibility Results. IACR Cryptology ePrint Archive 2012: 228 (2012)
9. Rafael Dowsley, Jörn Müller-Quade, Tobias Nilges: *Weakening the Isolation Assumption of Tamper-Proof Hardware Tokens.* ICITS 2015: 197-213
10. Yael Gertner, Sampath Kannan, Tal Malkin, Omer Reingold, Mahesh Viswanathan: *The Relationship between Public Key Encryption and Oblivious Transfer.* FOCS 2000: 325-335
11. Blaise Gassend, Dwaine Clarke, Marten van Dijk, Srinivas Devadas: *Silicon physical random functions.* ACM Conference on Computer and Communications Security 2002: 148-160
12. Blaise Gassend, Daihyun Lim, Dwaine Clarke, Marten v. Dijk, Srinivas Devadas: *Identification and authentication of integrated circuits.* Concurrency and Computation: Practice & Experience, pp. 1077 - 1098, Volume 16, Issue 11, September 2004.
13. Oded Goldreich: *Foundations of Cryptography: Volume II (Basic Applications).* Cambridge University Press, 2004.
14. Oded Goldreich, Silvio Micali, Avi Wigderson: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. STOC 1987: 218-229
15. Vipul Goyal, Yuval Ishai, Mohammad Mahmoody, Amit Sahai: *Interactive Locking, Zero-Knowledge PCPs, and Unconditional Cryptography.* Crypto 2010: 173-190. Full version available from IACR Cryptology ePrint Archive 2010: 89 (2010).
16. Jorge Guajardo, Sandeep S. Kumar, Geert Jan Schrijen, Pim Tuyls: *FPGA Intrinsic PUFs and Their Use for IP Protection.* CHES 2007: 63-80
17. Iftach Haitner, Eran Omri, Hila Zarosim: *Limits on the Usefulness of Random Oracles.* TCC 2013: 437-456
18. Clemens Helfmeier, Dmitry Nedospasov, Christian Boit, Jean-Pierre Seifert: *Cloning Physically Unclonable Functions.* HOST 2013.
19. Daniel E. Holcomb, Wayne P. Burleson, Kevin Fu: *Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers.* IEEE Trans. Computers, 2009.
20. Russell Impagliazzo, Steven Rudich: *Limits on the Provable Consequences of One-Way Permutations.* STOC 1989: 44-61

21. Joe Kilian: *Founding cryptography on oblivious transfer.* STOC (1988)
22. Stefan Katzenbeisser, Ünal Kocabas, Vincent van der Leest, Ahmad-Reza Sadeghi, Geert Jan Schrijen, Christian Wachsmann: J. Cryptographic Engineering 1(3): 177-186 (2011)
23. Sandeep S. Kumar, Jorge Guajardo, Roel Maes, Geert Jan Schrijen, Pim Tuyls: *The Butterfly PUF: Protecting IP on every FPGA.* HOST 2008: 67-70
24. Keith Lofstrom, W. Robert Daasch, Donald Taylor: *IC identification circuit using device mismatch.* Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International. IEEE, 2000.
25. Roel Maes: Physically Unclonable Functions – Constructions, Properties and Applications. Springer 2013, ISBN 978-3-642-41394-0, pp. 1-172.
26. Mohammad Mahmoody, Hemanta K. Maji, Manoj Prabhakaran: Limits of random oracles in secure computation. ITCS 2014: 23-34
27. Moni Naor: Bit Commitment Using Pseudorandomness. J. Cryptology 4(2): 151-158 (1991)
28. Rafail Ostrovsky, Alessandra Scafuro, Ivan Visconti, Akshay Wadia: *Universally Composable Secure Computation with (Malicious) Physically Uncloneable Functions.* Cryptology ePrint Archive, March 16, 2012.
29. Rafail Ostrovsky, Alessandra Scafuro, Ivan Visconti, Akshay Wadia: *Universally Composable Secure Computation with (Malicious) Physically Uncloneable Functions.* Eurocrypt 2013.
30. Ravikanth Pappu: *Physical One-Way Functions.* PhD Thesis, Massachusetts Institute of Technology, 2001.
31. Ravikanth Pappu, Ben Recht, Jason Taylor, Neil Gershenfeld: *Physical One-Way Functions*, Science, vol. 297, pp. 2026-2030, 20 September 2002.
32. Omer Reingold, Luca Trevisan, Salil P. Vadhan: *Notions of Reducibility between Cryptographic Primitives.* TCC 2004: 1-20.
33. Ulrich Rührmair: *Physical Turing Machines and the Formalization of Physical Cryptography.* Internal Manuscript, 2006. Put online on the IACR Cryptology ePrint Archive in 2011 as Report 188/2011.
34. Ulrich Rührmair: *Oblivious Transfer based on Physical Unclonable Functions (Extended Abstract).* TRUST Workshop on Secure Hardware, Berlin (Germany), June 22, 2010. Lecture Notes in Computer Science, Volume 6101, pp. 430 - 440. Springer, 2010.
35. Ulrich Rührmair, Heike Busch, Stefan Katzenbeisser: *Strong PUFs: Models, Constructions and Security Proofs.* In A.-R. Sadeghi, P. Tuyls (Editors): Towards Hardware Intrinsic Security: Foundation and Practice. Springer, 2010.
36. U. Rührmair, S. Devadas, F. Koushanfar: *Security based on Physical Unclonability and Disorder.* In M. Tehranipoor and C. Wang (Editors): "Introduction to Hardware Security and Trust". Springer, 2011.
37. Ulrich Rührmair, Daniel E. Holcomb: *PUFs at a glance.* DATE 2014: 1-6
38. Ulrich Rührmair, Marten van Dijk: *Practical Security Analysis of PUF-based Two-Player Protocols.* CHES 2012: 251-267.
39. Ulrich Rührmair, Marten van Dijk: *On the practical use of physical unclonable functions in oblivious transfer and bit commitment protocols.* J. Cryptographic Engineering 3(1): 17-28 (2013)
40. Ulrich Rührmair, Marten van Dijk: *PUFs in Security Protocols: Attack Models and Practical Security Evaluations.* IEEE Symposium on Security and Privacy (Oakland'13), 2013.
41. Ulrich Rührmair, Christian Hilgers, Sebastian Urban, Agnes Weiershäuser, Elias Dinter, Brigitte Forster, Christian Jirauschek: *Optical PUFs Revisited.* Cryptology ePrint Archive: Report 2013/215, 2013.
42. Ulrich Rührmair, Christian Jaeger, Michael Algasiner: *An Attack on PUF-based Session Key Exchange, and a Hardware-based Countermeasure: Erasable PUFs.* Financial Cryptography and Data Security 2011.
43. Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, Jürgen Schmidhuber: *Modeling Attacks on Physical Unclonable Functions.* ACM Conference on Computer and Communications Security, 2010.

44. George Savvides: *Interactive Hashing and reductions between Oblivious Transfer variants*. PhD Thesis, McGill University, 2007.
45. Daniel R. Simon: *Finding Collisions on a One-Way Street: Can Secure Hash Functions Be Based on General Assumptions?* EUROCRYPT 1998: 334-345.
46. Pim Tuyls, Boris Skoric: *Strong Authentication with Physical Unclonable Functions.* In: Security, Privacy and Trust in Modern Data Management, M. Petkovic, W. Jonker (Eds.), Springer, 2007.

APPENDIX

A A New PUF-based OT Protocol with Intended Security against Stateless PUFs-inside-PUFs Attacks (First Attempt, Which Fails)

In this appendix, we will deal with one straightforward attempt to use interleaved challenges to achieve security against stateless PUFs-inside-PUFs attacks. The following Protocol 6 is a straightforward combination of interleaved challenges and of the above Protocol 2 by Dachman-Soled et al. [4]. In order to prevent the PUFs-inside-PUFs attack of Section 5.3, k challenges are used not only to authenticate $\mathsf{PUF_S}$, but also in order to blind the secrets s_0 and s_1. This means that a cheating Receiver must know all of these k challenges in order to decrypt S_0 and S_1 and learn s_0 and s_1. The internal hash function of $\mathsf{PUF}^*_\mathsf{S}$ must hence take an output value of 1 for all of these k challenges (which makes it pass on the challenges to its internal pseudo-random function $\mathsf{PRF_2}$). The probability that this happens for all of the k challenges is exponentially small in k (recall that h would take value 1 with probability $\frac{1}{k+1}$).

Protocol 6

(A New OT-Protocol with Intended Security against PUFs-inside-PUFs Attacks — First Attempt, Which Fails)

PREPARATION PHASE:

1. Sender creates $\mathsf{PUF_S}$.
2. Sender collects k random, *"authenticating"* CRPs $(c_1^*, r_1^*), \ldots, (c_k^*, r_k^*)$ from $\mathsf{PUF_S}$.
3. Sender sends $\mathsf{PUF_S}$ to Receiver.
4. Receiver creates $\mathsf{PUF_R}$.
5. Receiver collects k random CRPs $(c_1, r_1), \ldots, (c_k, r_k)$ from the *"combined"* $\mathsf{PUF}_\oplus$ (where $\mathsf{PUF}_\oplus(c) := \mathsf{PUF_S}(c) \oplus \mathsf{PUF_R}(c)$, as before).
6. Receiver sends $\mathsf{PUF_S}$ and $\mathsf{PUF_R}$ to Sender.

OT-PHASE:

The Sender holds two bitstrings s_0, s_1. The receiver holds a choice bit b.

1. For $i = 1, \ldots, k$, Sender chooses random string pairs $x_0^i, x_1^i \in \{0,1\}^\lambda$, and sends all x_0^i, x_1^i to Receiver.
2. For $i = 1, \ldots, k$, Receiver defines

$$v_i := c_i \oplus x_b,$$

and sends all v_i to Sender.

3. For $i = 1, \ldots, k$, Sender defines
$$c_i^0 := v_i \oplus x_0^i, c_i^1 := v_i \oplus x_1^0.$$
4. *Challenge Mixing Step:*
   – Note that Sender now holds $3k$ challenges:
   $$c_1^*, \ldots, c_k^*, c_1^0, \ldots, c_k^0, c_1^1, \ldots, c_k^1.$$
   The c_i^* are all challenges to $\mathsf{PUF_S}$, while the c_i^0 and c_i^1 are all challenges to $\mathsf{PUF}_\oplus$.
   – The Sender applies these $3k$ challenges
   $$c_1^*, \ldots, c_k^*, c_1^0, \ldots, c_k^0, c_1^1, \ldots, c_k^1$$
   in **random order** to $\mathsf{PUF_S}$ or to $\mathsf{PUF}_\oplus$, respectively, collecting (again in random order) the $3k$ corresponding responses
   $$r_1^*, \ldots, r_k^*, r_1^0, \ldots, r_k^0, r_1^1, \ldots, r_k^1.$$
   Note again that the r_i^* are responses of $\mathsf{PUF_S}$, while the r_i^0 and r_i^1 are responses of $\mathsf{PUF}_\oplus$.
5. *Authentication Step:*
   – Sender compares the obtained responses $r_1^*, \ldots, r_k^*$ to the responses of the precorded *"authenticating"* CRPs from the preparation phase.
   – If they match, he continues. Else, he aborts.
6. Sender computes
$$S_0 := s_0 \oplus r_1^{b'} \oplus \ldots \oplus r_k^{b'},$$
and
$$S_1 := s_1 \oplus r_1^{1-b'} \oplus \ldots \oplus r_k^{1-b'},$$
and sends S_0, S_1 to Receiver.
7. Receiver produces as output $s_b := S_b \oplus r_1 \oplus \ldots \oplus r_k$.

A Subtle Attack. The protocol seems secure at first sight, but there is a very subtle, yet efficient attack. The key observations are:

– A malicious Receiver can strongly influence the choice of the challenges c_i^b and c_i^{1-b} by deviating from the protocol, and by sending selected values v_i in Step 2 of the protocol. It is not too difficult to see that by choosing the v_i maliciously, he can in fact enforce the challenges c_i^b and c_i^{1-b} to take any values that to fulfill the side condition that
$$c_i^b \oplus c_i^{1-b} = x_i^0 \oplus x_i^1.$$
This leaves a substantial degree of freedom for the Receiver to enforce values for c_i^b and c_i^{1-b} that enable certain attacks. [11]

[11] Interestingly, the same observation has also been at the heart of earlier attacks on the PUF-based OT-protocol of Brzuska et al. [1], which were published at CHES 2012 and in JCEN 2013 by Rührmair and van Dijk [38, 39].

– If the challenges c_i^b and c_i^{1-b} are influenced maliciously by the Receiver (and are later applied to the bad PUF by the unknowing Sender in the course of the protocol), they can communicate a “message” or some kind of helpful “information” to the bad PUF.
– More concretely, in the above Protocol 6, a cheating Receiver could enforce that all of the challenges c_i^b have a first half that consists of only zeros, and that all of the challenges c_i^{1-b} have a second half that consists of only zeros. This choice does not hinder or violate the abovementioned condition that $c_i^b \oplus c_i^{1-b} = x_i^0 \oplus x_i^1$.

These observations lead to the following attack: A cheating Receiver substitutes the original PUF_S that he had received in the preparation phase of the protocol. He replaces it by a stateless malicious $\mathsf{PUF}_\mathsf{S}^*$ that encapsulates PUF_S, similar to the construction in Figure 3. This $\mathsf{PUF}_\mathsf{S}^*$ is programmed to have the following functionality:

– It recognizes all mulitbit strings c_i^b and c_i^{1-b} by the fact that they start or end with a half that solely conists of zeros (see above).
– Having recognized all c_i^b and c_i^{1-b} in this manner, $\mathsf{PUF}_\mathsf{S}^*$ can immediately deduce which of the applied challenges are the authenticating challenges c_i^* (it are those which do not have a first or second half consisting solely of zeros.)
– The interface of $\mathsf{PUF}_\mathsf{S}^*$ passes on all authenticating challenges c_i^* to the internal PUF_S that $\mathsf{PUF}_\mathsf{S}^*$ encapsulates, and presents the responses of PUF_S to the outside. This allows $\mathsf{PUF}_\mathsf{S}^*$ to survive the authentication unrecognizedly.
– All other applied challenges c_i^b and c_i^{1-b} are passed on by the interface of $\mathsf{PUF}_\mathsf{S}^*$ to a simulatable bad PUF, for example to a pseudo-random function (PRF), which is simulatable by the adversary (see again Figure 3).

Using such a *stateless* malicious $\mathsf{PUF}_\mathsf{S}^*$, the cheating Receiver remains unnoticed despite our authentication step and interleaved challenges. Still, he can (with overwhelming probability) simulate and learn all of the responses $r_1^0, \ldots, r_k^0, r_1^1, \ldots, r_k^1$. (Such simulation will only then *not* be possible if by coincidence, one of the challenges $c_1^0, \ldots, c_k^0, c_1^1, \ldots, c_k^1$ will start or end with a half of solely zeros; but this happens only with exponentially small probability in λ, where λ is the length of the challenges.) Knowing all $r_1^0, \ldots, r_k^0, r_1^1, \ldots, r_k^1$ will allow the Receiver to decode and learn both s_0 and s_1, however, breaking the confidentiality of the Sender. This breaks Protocol 6 with a probability that is exponentially close to 1 in λ.

The above attack works since the two sets of challenges c_i^0 and c_i^1 can be influenced by the Receiver. We therefore had to think about ways how this can be prevented; this was our motivation behind using interactive hashing (IH) in Protocol 5. IH had already been used by the historically first PUF protocol for OT by Rührmair in 2010 [34], and in subsequently improved versions by Rührmair and van Dijk [38, 39], for exactly the same reasons.