Moderately Hard Functions: Definition, Instantiations, and Applications*

Joël Alwen[1] and Björn Tackmann[2]

[1] IST, Vienna, Austria, jalwen@ist.ac.at
[2] IBM Research – Zurich, Switzerland, bta@zurich.ibm.com

Abstract. Several cryptographic schemes and applications are based on functions that are both reasonably efficient to compute and moderately hard to invert, including client puzzles for Denial-of-Service protection, password protection via salted hashes, or recent proof-of-work blockchain systems. Despite their wide use, a definition of this concept has not yet been distilled and formalized explicitly. Instead, either the applications are proven directly based on the assumptions underlying the function, or some property of the function is proven, but the security of the application is argued only informally. The goal of this work is to provide a (universal) definition that decouples the efforts of designing new moderately hard functions and of building protocols based on them, serving as an interface between the two.
On a technical level, beyond the mentioned definitions, we instantiate the model for four different notions of hardness. We extend the work of Alwen and Serbinenko (STOC 2015) by providing a general tool for proving security for the first notion of memory-hard functions that allows for provably secure applications. The tool allows us to recover all of the graph-theoretic techniques developed for proving security under the older, non-composable, notion of security used by Alwen and Serbinenko.
As an application of our definition of moderately hard functions, we prove the security of two different schemes for proofs of effort (PoE). We also formalize and instantiate the concept of a non-interactive proof of effort (niPoE), in which the proof is not bound to a particular communication context but rather any bit-string chosen by the prover.

1 Introduction

Several cryptographic schemes and applications are based on (computational) problems that are "moderately hard" to solve. One example is hashing passwords with a salted, moderately hard-to-compute hash function and storing the hash in the password file of a login server. Should the password file become exposed through an attack, the increased hardness of the hash function relative to a standard one increases the effort that the attacker has to spend to recover the passwords in a brute-force attack [65,68,44]. Another widely-cited example of this approach originates in the work of Dwork and Naor [38], who suggested the

* The proceedings version is in TCC 2017. This is the full version.

use of a so-called *pricing function*, supposedly moderately hard to compute, as a countermeasure for junk mail: the sender of a mail must compute a moderately hard function (MoHF) on an input that includes the sender, the receiver, and the mail body, and send the function value together with the message, as otherwise the receiver will not accept the mail. This can be viewed as a proof of effort[3] (PoE), which, in a nutshell, is a 2-party (interactive) proof system where the verifier accepts if and only if the prover has exerted a moderate amount of effort during the execution of the protocol. Such a PoE can be used to meter access to a valuable resource like, in the case of [38], the attention of a mail receiver. As observed by the authors, requiring this additional effort would introduce a significant obstacle to any spammer wishing to flood many receivers with unsolicited mails. Security was argued only informally in the original work. A line of follow-up papers [1,37,39] provides a formal treatment and proves security for protocols that are intuitively based on functions that are moderately hard to compute on architectures with limited cache size.

PoEs have many applications beyond combatting spam mail. One widely discussed special case of PoE protocols are so-called cryptographic puzzles (or client puzzles, e.g. [54,14,71,27,48,29]), which are mainly targeted at protecting Internet servers from Denial-of-Service attacks by having the client solve the puzzle before the server engages in any costly operation. These PoEs have the special form of consisting of a single pair of challenge and response messages (i.e., one round of communication), and are mostly based on either inverting a MoHF [54], or finding an input to an MoHF that leads to an output that has a certain number of trailing zeroes [2]. More recently, crypto-currencies based on distributed transaction ledgers that are managed through a consensus protocol based on PoEs have emerged, most prominently Bitcoin [66] and Ethereum [24], and are again based on MoHFs. In a nutshell, to append a block of transactions to the ledger, a so-called *miner* has to legitimate the block by a PoE, and as long as miners that control a majority of a computing power are honest, the ledger remains consistent [45].

The notions of hardness underlying the MoHFs that have been designed for the above applications vary widely. The earliest and still most common one is computational hardness in terms of the number of computation steps that have to be spent to solve the problem [38,54,29,66]. Other proposals exploit the limited size of fast cache in current architectures and are aimed at forcing the processor to access the slower main memory [1,37,39], the use of large amounts of memory during the evaluation of the function [68,44,11], or even disk space [40].

Given the plethora of work (implicitly or explicitly) designing and using MoHFs, one question soon comes to mind: is it possible to use the MoHF designed in one work in the application context of another? The current answer is sobering. Either the security notion for the MoHF is not quite sufficient for proving

[3] We intentionally use the term *effort* instead of *work* since the latter is often associated with computational work, while a MoHF in our framework may be based on spending other types of resources such as memory.

the security of the targeted applications. Or security of the application is proven directly without separating out the properties used from the underlying MoHF.

For example, in the domain of memory-hard functions (an increasingly common type of MoHF first motivated by Percival in [68]) the security of MoHF applications is generally argued only informally. Indeed, this likely stems from the fact that proposed definitions seem inadequate for the task. As argued by Alwen and Serbinenko [11], the hardness notion used by Percival [68] and Forler *et al.* [44] is not sufficient in practical settings because it disregards that an attacker may amortize the effort over multiple evaluations of the function.[4] Yet the definition of [11], while preventing amortization, is also not (known to be) useful in proving the security of higher-level protocols, because it requires high average-case, instead of worst-case, complexity. Worse, like all other MoHF definitions in the literature (e.g. [19,4]), it focuses only on the hardness of *evaluating* the function. However, all applications present the adversary with the task of *inverting* the MoHF in same form or another.[5]

In other areas, where the application security *is* explicitly proven [1,37,39], this is done directly with out separating out the properties of the underlying MoHF. This means that (a) the MoHF (security) cannot easily be "extracted" from the paper and used in other contexts, and (b) the protocols cannot easily be instantiated with other MoHFs. Furthermore, the security definitions come with a hard-wired notion of hardness, so it is *a priori* even more difficult to replace the in-built MoHF with one for a different type of hardness.

Consequently, as already discussed by Naor in his 2003 invited lecture [67], what is needed is a unifying theory of MoHFs. The contribution of this paper is a step toward this direction. Our goal is to design an abstract notion of MoHF that is flexible enough to model various types of functions for various hardness notions considered in the literature, but still expressive enough to be useful in a wide range of applications. We propose such a definition, show (with varying degrees of formality) that existing constructions for various types of hardness instantiate it, and show how it can be used in various application scenarios. Not all proof-of-work schemes, however, fall into the mold of the ones covered in this work. For example the recently popular Equihash [20] has a different form.[6]

[4] Additionally [44] do not consider an attack with parallel computational capabilities such as a circuit.

[5] The problem, at least with respect to proving security of applications is that, for almost any reasonable notion of complexity and function f we could modify f to prepend its input to its output to obtain a new function with just as high complexity as the original, but which is trivial to invert.

[6] Nevertheless, we conjecture that Equihash could also be analyzed in out framework. In particular, if we can always model the underlying hash function used by Equihash as a (trivially secure) MoHF. Then, by assuming the optimality of Wagner's collision finding algorithm (as done in [20]) one could compute the parameters for which Equihash gives rise to our proof-of-effort definition in Section 7. We leave this line of reasoning for future work.

Outline. In Section 2, we provide an extended abstract that contains a self-contained though mostly semi-formal treatment of the results in this paper. The section intentionally follows the structure of the main technical sections, so that the interested reader can easily find the detailed material there. Section 3 contains the preliminaries for the full, technical part. In Section 4, we describe our new definition of MoHF, which is already sketched in Section 2, more formally. Next, in Section 5, we instantiate the MoHF definition for the case of memory-hard functions, and in Section 6 we discuss other types of moderately hard functions from the literature. In Section 7, we describe a (composable) security definition for PoE. In Section 8, we then continue to describing an analogous definition for a non-interactive proof of effort (niPoE), and again give an instantiation based on hash trail. In Section 9, we discuss the composition of the MoHF definition and the PoE and niPoE applications more concretely. A detailed discussion of related work beyond the overview above, and a discussion of the applicability of our results to the scenarios discussed from previous literature, appear in Section 10. Appendices contain additional material (Appendices A and C) and proofs (Appendix B).

2 Extended abstract

2.1 Moderately hard functions

The starting point of our MoHF definition is the observation that many works in this area seems to—more or less implicitly—assume that a MoHF behaves like a random oracle. For instance, the hash-trail scheme [2] that has been widely used [66,49] works under the assumption that the outputs of the function are uniform, and papers proving memory-hardness generally prove hardness in the forward direction [68,44,11], although most applications actually need hardness in the backward direction. This appears to implicitly assume that forward-hardness implies backward-hardness, which is somewhat true for random oracles, but false in general. The immediate idea would then be to require that a MoHF be indifferentiable from a random oracle [64], but that alone would not be sufficient as plain indifferentiability does not properly model resource consumption [72,34], which however is central to the notion of MoHF. The core idea of our definition can be seen as making the indifferentiability framework "resource-aware."

We recall that the goal of indifferentiability can be seen as defining when a protocol or scheme π, which may use a certain resource[7] $\mathcal{S}$, provides the guarantees specified by means of some (often idealized) resource $\mathcal{T}$. The resources $\mathcal{S}$ and $\mathcal{T}$ each provide two different interfaces via which they can be accessed: a *private* interface "priv" that defines how an honest party can access the resource, and a *public* interface "pub" that specifies the capabilities of an adversary. On a high level, and as depicted in Figure 1, π realizes $\mathcal{T}$ from $\mathcal{S}$ in the sense of indifferentiability if there exists a simulator σ such that the two settings depicted in Figure 1 are indistinguishable.

[7] All resources here are described as discrete reactive systems that respond to queries from their environment, see [26,69,60] for instantiations of this concept.

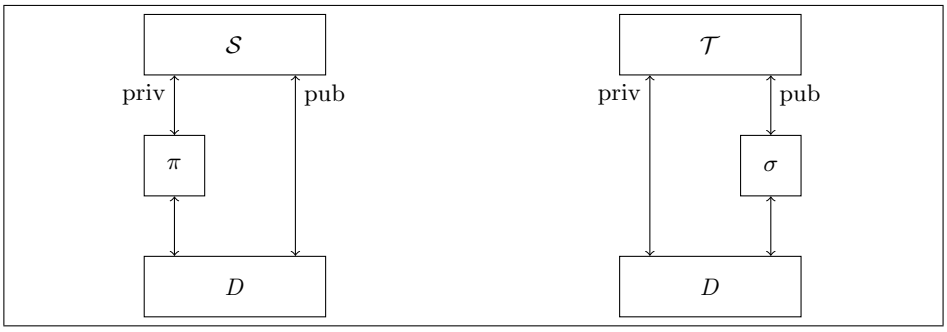


Fig. 1: Indifferentiability. **Left:** Distinguisher D connected to protocol π using the priv-interface of the real-world resource $\mathcal{S}$, denoted $D\left(\pi^{\text{priv}}\mathcal{S}\right)$. **Right:** Distinguisher D connected to simulator σ attached to the pub-interface of the ideal-world resource $\mathcal{T}$, denoted $D\left(\sigma^{\text{pub}}\mathcal{T}\right)$.

The ideal-world MoHF. We model an ideal MoHF as a restricted random oracle (RRO) to which only a limited number of queries can be made. More precisely, we write $\mathcal{T}^{\text{RRO}}_{\mathsf{a},\mathsf{b}}$ to denote such a random oracle allowing $\mathsf{a} \in \mathbb{N}$ queries to the honest party and $\mathsf{b} \in \mathbb{N}$ queries to the adversary. We then require that a candidate function being evaluated in a setting with bounded resources be indifferentiable from this idealization. This approach has the advantage of being extremely convenient in the analysis of higher-level protocols based on MoHFs: the analysis can be performed in the restricted random-oracle model, under consideration of the bounds a, b on the number of queries, and the composability of indifferentiability guarantees that the result carries over to the setting with the real MoHF.

In particular, our definition implies one-wayness for the MoHF because the random oracle is one-way (and therefore an efficient inverter can easily be used turned into a distinguisher in the indifferentiability statement).

The real-world setting. The real-world setting models the intuition that the resources of both the honest party and the adversary are bounded. For each notion of hardness, we have to describe the appropriate computational model and complexity measure; in particular, for each hardness notion we will have a different description of the real world.

Generally, the real world is described in terms of a computation resource which allows both the honest party and the adversary to input an algorithm, and to subsequently provide (possibly repeatedly) inputs on which the algorithm will be evaluated. The computational resource is bounded in the sense that evaluating the algorithm incurs some cost, and depending on the cost of each evaluation and the bounds imposed by the resource, each algorithm may only be executed a certain number of times, or maybe not at all. Complexity measures may charge, e.g., for certain operations of the algorithm (such as each call to the random oracle), for the memory use during the computation, for each memory access, and so on.

In all cases we consider, the real-world resource will provide the algorithms with a random oracle that can be queried. That means that the function implemented by the algorithm is actually an *oracle function*, a term that we specify in Definition 4. We usually denote such functions, which are defined relative to an oracle, as $f^{(\cdot)}$.

For several complexity measures (such as for memory hardness; our main technical result) we restrict the adversary allowing it to input just one algorithm and to evaluate this algorithm on only one input. At first glance, this restriction may appear needlessly restrictive, and therefore calls for further discussion. *(When) is it necessary?* For complexity measures such as memory hardness or memory-boundedness, continuous interaction between the algorithm and the environment would allow the algorithm to "outsource" memory to the environment, essentially circumventing the resource bound. We stress that previous, stand-alone definitions of memory-hard functions have the same restriction, it only appears less explicitly there. For other complexity measures such as counting the oracle calls of an random-oracle algorithm, such a restriction is not required as the environment is not given direct access to the random oracle. *(When) is it natural?* The purpose of our definition is to exactly specify the (assumed) resource bounds per party. In applications of MoHFs, there is often a natural beginning at which the adversary, e.g., receives a challenge, and begins the computation. The resource-consumption of this computation is exactly what we want to measure. Statements about moderately hard functions inherently relate to the computational resources available to the adversary. *Why not consider bounded distinguishers?* The purpose of the distinguisher is to model the environment in which the protocol is used; in a sense, the distinguisher can be understood as comprising the entire remaining world, including all other cryptographic protocols possibly executed concurrently. Bounding the distinguisher would therefore result in too-weak security statements, since reasonably large bounds render the statement about the MoHF imprecise, while smaller bounds prevent concurrent composition with other protocols. *Why is only one algorithm input at* pub*?* Allowing multiple algorithms to be input generally allows outsourcing to the distinguisher (as above). In many practical applications, different sessions of the protocol separate instances of the underlying random oracle by, e.g., salting, and we treat each *salted* random oracle as an individual instance. We leave it to future work to make this argument formal, for example using the techniques of [80,36].

Definition and the use in applications. Given the above descriptions of the resources, our security definition is simply based on indifferentiability of the MoHF in the described real-world setting from the query-bounded random oracle. In the real-world setting, the honest party uses the prescribed algorithm and repeatedly evaluates it on inputs adaptively chosen by the distinguisher, whereas the adversarial party uses the algorithm prescribed by the distinguisher. In the ideal-world setting, both parties access the query-bounded random oracle; the simulator at the pub-interface takes as input the algorithm from the distinguisher and evaluates it.

Definition 1 (MoHF security, informal). *Let $f^{(\cdot)}$ be an oracle function and* $\mathtt{naïve}$ *be an algorithm for computing $f^{(\cdot)}$. For non-negative* $\mathtt{a}, \mathtt{b}$ *and ε a computational resource $\mathcal{S}$ describing the real world, $(f^{(\cdot)}, \mathtt{naïve})$ is a* $(\mathtt{a}, \mathtt{b}, \varepsilon)$-secure moderately hard function *if there is a simulator σ such that $\pi^{\text{priv}}_{\mathtt{naïve}}\ \mathcal{S}$, where $\pi_{\mathtt{naïve}}$ is the protocol that inputs algorithm* $\mathtt{naïve}$ *into the resource and then evaluates it on all inputs provided by the environment, is ε-indistinguishable from $\sigma^{\text{pub}}\ \mathcal{T}^{\text{RRO}}_{\mathtt{a},\mathtt{b}}$.*

Despite the fact that indifferentiability-based definitions are generally composable, previous results have shown that the interpretation of the composed statements requires care especially in resource-restricted scenarios [72,34].

As we further discuss in Sections 7, 8, and 10, our definition is useful in several application areas of MoHFs where we would like to limit the rate at which an adversary can evaluate a critical function while simultaneously still allowing an honest (potentially weaker) party from still evaluating said function at a moderate rate. Examples include functions in password-based cryptography such as Key Derivation Functions and password hashing algorithms. Another set of examples are proofs of effort and non-interactive proofs of effort. Generally, these are applications where for different instances of the application disjoint inputs to the MoHF are used, and we can therefore consider these applications as using multiple instances of our resources. (Salted password storage is another example of such a use case.) On the other hand, the assumptions formalized in the real-world model are not compatible with all use cases and applications of MoHFs. For instance, using our definition of MoHF in the blockchain model of Garay *et al.* [45] does not work, since the adversary in that model can invoke the same algorithm multiple times, and these invocations necessarily involve *the same instance* of the MoHF.

Overall, one main advantage of our definition is that the idealization is independent of a concrete hardness notion, and applications can simply be analyzed in the query-bounded random oracle model, which makes the protocol analysis both more general and easier to understand. The generic composition theorem for indifferentiability allows to combine any application (for which our definition of MoHF leads to meaningful statements, see above) with any MoHF, and in particular with all types of MoHF described in the subsequent two sub-sections.

2.2 Memory-hard functions

Next, in Section 5, we instantiate the MoHF definition for the case of *memory-hard function* (MHF). Intuitively, an MHF is a function such that (1) the most efficient strategy for inverting the MHF on low entropy inputs is a guess-and-check strategy and (2) the product of the area and time of any parallel device (amortized per instance of the MHF the device evaluates) is large. This complexity notion is called (amortized) area-time (aAT) complexity [4] and has origins in the domain of VLSI [78]. In the formalization of MHFs in this work we actually only consider the area of the memory components. (Clearly our lower bounds on

memory-area are also lower bounds on the total area.[8]) Still, the techniques we introduce here carry over to the more complicated setting where the area used for computational logic is also considered.

A bit more precisely we formalize MHFs using a variant of the Random Oracle Model (pROM). That is we consider a resource-bounded computational device $\mathcal{S}$ with a priv- and a pub-interface capturing the pROM (adapted from [9]). Let $w \in \mathbb{N}$. Upon startup, $\mathcal{S}^{w\text{-PROM}}$ samples a fresh random oracle $h \leftarrow\!\!\$\ \mathbb{H}_w$ with range $\{0,1\}^w$. Now, on both interfaces, $\mathcal{S}^{w\text{-PROM}}$ accepts as input a pROM algorithm $\mathtt{A}$ which is an oracle algorithm with the following behavior.

A *state* is a pair $(\tau, \mathbf{s})$ where *data* τ is a string and $\mathbf{s}$ is a tuple of strings. The output of step i of algorithm $\mathtt{A}$ is an *output state* $\bar{\sigma}_i = (\tau_i, \mathbf{q}_i)$ where $\mathbf{q}_i = [q_i^1, \ldots, q_i^{z_i}]$ is a tuple of *queries* to h. As input to step $i+1$, algorithm $\mathtt{A}$ is given the corresponding *input state* $\sigma_i = (\tau_i, h(\mathbf{q}_i))$, where $h(\mathbf{q}_i) = [h(q_i^1), \ldots, h(q_i^{z_i})]$ is the tuple of *responses* from h to the queries $\mathbf{q}_i$. In particular, for a given h and random coins of $\mathtt{A}$, the input state σ_{i+1} is a function of the input state σ_i. The initial state σ_0 is empty and the input x_{in} to the computation is given a special input in step 1.

For a given execution of a pROM, we are interested in the following complexity measure. We denote the bit-length of a string s by $|s|$. The *length* of a state $\sigma = (\tau, \mathbf{s})$ with $\mathbf{s} = (s^1, s^2, \ldots, s^y)$ is $|\sigma| = |\tau| + \sum_{i \in [y]} |s^i|$. The *cumulative memory complexity* (CMC) of an execution is the sum of the lengths of the states in the execution. More precisely, let us consider an execution of algorithm $\mathtt{A}$ on input x_{in} using coins \$ with oracle h resulting in $z \in \mathbb{Z}_{\geq 0}$ input states $\sigma_1, \ldots, \sigma_z$, where $\sigma_i = (\tau_i, \mathbf{s}_i)$ and $\mathbf{s}_i = (s_i^1, s_i^2, \ldots, s_i^{y_j})$. Then the *cumulative memory complexity* (CMC) of the execution is

$$\mathsf{cmc}(\mathtt{A}^h(x_{\mathsf{in}}; \$)) = \sum_{i \in [z]} |\sigma_i| \ ,$$

while the *total number of RO calls* is $\sum_{i \in [z]} y_j$. More generally, the CMC (and total number of RO calls) of several executions is the sum of the CMC (and total RO calls) of the individual executions.

The device $\mathcal{S}^{w\text{-PROM}}$ imposes constraints on the algorithm it executes. In particular for an implicit pair of positive integers $(q_{\text{priv}}, m_{\text{priv}})$ the pROM algorithm on the priv interface is allowed to make a total of q_{priv} RO calls and use CMC at most m_{priv} summed up across all of the algorithms executions. Similar constraints hold on the pub interface for implicit parameters $(q_{\text{pub}}, m_{\text{pub}})$.

As usual for memory-hard functions, to ensure that the honest algorithm can be run on realistic devices, $\mathcal{S}^{w\text{-PROM}}$ restricts the algorithms on the priv-interface to be *sequential*. That is, the algorithms can make only a single call to h per step. Technically, in any execution, for any step j it must be that $y_j \leq 1$. No such restriction is placed on the adversarial algorithm reflecting the power (potentially) available to such a highly parallel device as an ASIC.

[8] For further explanation as to why this lower bound is likely quite tight for all MHFs we consider we refer to [11].

We conclude the section with the formal definition of a memory-hard function in the pROM. The definition is a particular instance of and MoHF defined in Definition 5 formulated in terms of exact security.

Definition 2 ((Parallel) memory-hard function, informal). *Let $f^{(\cdot)}$ be an oracle function and* naïve *be a pROM algorithm for computing $f^{(\cdot)}$. Consider the function families:*

$$\mathsf{a} = \{\mathsf{a}_w \in \mathbb{N}\}_{w\in\mathbb{N}}\ , \qquad \mathsf{b} = \{\mathsf{b}_w \in \mathbb{N}\}_{w\in\mathbb{N}}\ , \qquad \epsilon = \{\epsilon_w \in \mathbb{R}_{\geq 0}\}_{w\in\mathbb{N}}\ .$$

Then $F = (f^{(\cdot)}, \mathtt{naïve})$ is called an $(\mathsf{a}, \mathsf{b}, \epsilon)$-memory-hard function *(MHF) if $\forall w \in \mathbb{N}$ F is an $(\mathsf{a}_w, \mathsf{b}_w, \epsilon_w)$-secure moderately hard function family for $\mathcal{S}^{w\text{-PROM}}$.*

Constructing MHFs. Now that we have defined MHFs we turn to their construction. To this end, we prove the main result of this section; a theorem bounding the parameters a, b and ϵ in Definition 2 for a given *graph function* (a.k.a. hash graph). Graph functions are a class of functions over bit strings constructed by fixing a mode of operation over a round function. In practice, most round functions are derived from the compression functions of cryptographic hash functions such as Blake2b [18]. As such, in this work the round function is modeled as a (fixed input length) random oracle (RO) also called an ideal compression function. We view the mode of operation as being given by a (constant in-degree) directed acyclic graph (DAG) which describes how the inputs and outputs of the calls to the RO are related to each other. First introduced in [39], graph functions have since appeared in many works on somewhat-hard-to-compute functions (e.g. [11,7,42,41,40,44,18,23,6] and many others) for various concrete notions of "hard to compute".

Let f_G be a graph function with a mode of operation over a RO given by G. The main theorem of this section (Theorem 4) quantifies parameters a b and ϵ for which f_G is an MHF in term of the *cumulative pebbling complexity* (CPC) of G [11]). The CPC of G is a complexity measure given in terms of simple pebbling game over DAGs. (c.f. Definition 8 and 9.) This has several consequences. For starters Theorem 4 reduces the goal of constructing new MHFs satisfying our comparatively strong definition to finding new practical graphs with high CPC (e.g. [6]. What makes the result particularly useful though is that CPC is already a relatively well-understood property (in particular for the graphs underlying many prominent candidate memory-hard functions in the literature [4,7,10]). Thus Theorem 4, together with [11,7] immediately results in security proofs for several proposed graph functions in the new MoHF framework (including for Argon2 [18], the most widely deployed graph function based MHF in practice).

2.3 Other types of MoHFs

In Section 6, we instantiate the MoHF definition for other types of moderately hard functions from the literature. In particular, we briefly review weak memory-hard functions and memory-bound functions, and shortly discuss one-time computable functions and uncomputable functions.

Weak memory-hard functions. A class of MoHFs considered in the literature that are closely related to MoHFs are *weak* MoHFs. Intuitively, they differ from MoHFs only in that they also restrict adversaries to being sequential.[9] On the one hand, it may be easier to construct such functions compared to full blown MoHF. In fact, for the data-independent variant of MoHFs, [4] proves that a graph function based on a DAG of size n always has cmc of $O(wn^2/\log(n))$ (ignoring $\log\log$ factors). Yet, as discussed below, the results of [56,44] and those described below show that we can build W-MoHFs from similar DAGs with sequential cmc of $\mathcal{O}(2n^2)$. Put differently, W-MoHFs allow for strictly more memory consumption per call to the RO than is possible with MoHFs. This is valuable since the limiting factor for an adversary is often the memory consumption while the cost for honest parties to enforce high memory consumption is the number of calls they must perform to the RO.

We capture weak MoHFs in the MoHF framework by restricting the real world resource-bounded computational device $\mathcal{S}^{w\text{-SROM}}$ to the *sequential random oracle model* (sROM). Given this definition we can now easily adapt the pebbling reduction of Theorem 4 to obtain a tool for constructing W-MoHFs, which has some immediate implications. In [56], Lengaur and Tarjan prove that the DAGs underlying the two graph functions Catena Dragonfly and Butterfly [44] have $\mathsf{scpc} = \mathcal{O}(n^2)$. In [44], the authors extend these results to analyze the scpc of stacks of these DAGs. By combining those results with the pebbling reduction for the sROM, we obtain parameters $(\mathsf{a}, \mathsf{b}, \epsilon)$ for which the Catena functions are provably W-MoHFs. Similar implications hold for the pebbling analysis done for the Balloon Hashing function in [23].

Memory-bound functions. Another important notion of MoHF from the literature has been considered in [37,39]. These predate MHFs and are based on the observation that while computation speeds vary greatly across real-world computational devices, this is much less so for memory-access speeds. Under the assumption that time spent on a computation correlates with the monetary cost of the computation, this observation motivates measuring the cost of a given execution by the number of cache misses (i.e., memory accesses) made during the computation. A function that requires a large number of misses, regardless of the algorithm used to evaluate the function, is called a *memory-bound* function.

To formalize memory-bound functions in the MoHF framework, we describe the real-world resource-bounded computational device $\mathcal{S}^{w\text{-MB}}$. It makes use of RO with w-bits of output and is parametrized by a description of the memory hierarchy. In [37,39] the authors describe such functions (with several parameters each) and prove that the hash-trail construction applied to these functions results in a PoE for a notion of "effort" captured by memory-boundness. We conjecture that the proofs in those works carry over to the notion of memory-bound MoHFs described above (using some of the techniques at the end of the proof of Theorem 4). Yet, we believe that a more general pebbling reduction (similar

[9] If the adversary is restricted to using general-purpose CPUs and not ASICs or FPGAs with their massive parallelism, this restriction may be reasonable.

to Theorem 4) is possible for the above definition. Such a theorem would allow us to construct new and improved memory-bound functions.

One-time computable and uncomputable functions. Another—less widely used—notion of MoHFs appearing in the literature are *one-time computable* functions [42]. Intuitively, these are sets of T pseudo-random functions (PRFs) $f_1, \ldots, f_T$ with long keys (where T is an *a priori* fixed, arbitrary number). An honest party can evaluate each function f_i exactly once, using a device with limited memory containing these keys. On such a device, evaluating the i^{th} PRF provably requires deleting all of the first i keys. Therefore, if an adversary (with arbitrary memory and computational power) can only learn a limited amount of information about the internal state of the device, then regardless of the computation performed on the device, the adversary will never learn more than one input/output pair per PRF. The authors describe the intuitive application of a password-storage device secure against dictionary attacks. An advantage of using the MoHF framework to capture one-time computable functions could be proving security for such an application (using the framework's composition theorem).

We describe a model for one-time computable functions and uncomputable functions in Section 6, where we also sketch a new (hypothetical) application for one-time computable functions in the context of anonymous digital payment systems.

2.4 Interactive proofs of effort

In Section 7, we describe a composable security definition for PoE. That means, as in the definition of MoHF above, we describe an (idealized) resource that models the guarantee that using a PoE provides to higher-level protocols. We then prove a condition that is analogous to indifferentiability, namely that there exists a simulator for the attacker, such that using the protocol on the assumed real-world resources is indistinguishable from the setting described by the idealized resource and the simulator. One of the assumed resources of our PoE will be the query-bounded random oracle that can then be instantiated using any MoHF that satisfies our definition.

The proof-of-effort resource. The high-level guarantees provided by a PoE to higher-level protocols can be described as follows. Prover P and verifier V interact in some number $n \in \mathbb{N}$ of sessions, and in each of the sessions verifier V expects to be "convinced" by prover P's spending of effort. Prover P can decide how to distribute the available resources toward convincing verifier V over the individual sessions; if prover P does not have sufficient resources to succeed in all sessions, then P can distribute its effort over the sessions. Verifier V's protocol provides as output a bit that is 1 in all sessions where the prover attributed sufficient resources, and 0 otherwise.

We formalize these guarantees in a resource that provides to P and V some number n of sessions in which they interact, and that is parametrized by a

function $\phi : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and a number $a \in \mathbb{N}$. In each of the n sessions, prover P can repeatedly attempt to convince verifier V; the probability for this to succeed in the i^{th} attempt within a session is $\phi(i)$, and P will learn whether the attempt was successful. Accumulated over all sessions, P has a attempts before the resource stops accepting further inputs. Verifier V can repeatedly test in each session whether P's proof succeeded. For a dishonest prover, the interface provided by the resource is slightly different, but only in the sense that after solving in a session, the result has to be made available to V explicitly. (This models the fact that a dishonest prover may solve the problem but then send a wrong solution to the verifier.) This intuitively described resource is made formal in Section 7.

We then prove, for two different schemes, one based on inverting the MoHF on a particular value, and one based on the hash trail scheme [2], that the described resource is indeed constructed in a setting where an MoHF, modeled as query-bounded random oracle, is available.

The hash-inversion scheme. The first scheme we consider is based on requiring the prover to invert the MoHF on a chosen distribution—the choice of this distribution also specifies the hardness of the problem [54]. Let f be a MoHF with range R and domain $\mathbb{N} \times D'$, and $D \subseteq D'$; for simplicity we assume that the distribution over D is uniform.

In each session $i \in \{1, \ldots, n\}$, verifier V samples a value x_i from D and sends $y_i = f(i, x_i)$ to P. In each session i, prover P then enumerates all $x \in D$ in some order and checks whether $f(i, x) = y_i$; in that case, x is a valid solution for session i, and is therefore sent to V. Verifier V checks the correctness of the solution, but accept only one attempt per session. We then prove the following theorem.

Theorem 1 (informal). *Define $\zeta_j \coloneqq (|D| - j + 1)^{-1}$. If V can evaluate f at least $2n$ times, then the described protocol is a proof of effort parametrized by $\phi : j \mapsto \zeta_j + \frac{1-\zeta_j}{|R|}$, and where the number a of attempts for the verifier is the same as the prover's bound in the query-bounded random oracle (for honest provers) resp. larger by n (for dishonest provers).*

The additional n attempts for the dishonest prover stem from the fact that sending an attempt $\tilde{x} \in D$ does not require evaluating the MoHF on the input, but such a guess is only possible once in each one of the n sessions. Verifier V has to evaluate f twice per session, once to generate the challenge and once to verify the solution.

The hash-trail scheme. The second scheme is based on requiring the prover to provide an input to the MoHF that leads to an output of some form, such as some number $d \in \mathbb{N}$ of trailing zero bits [2]. We consider a MoHF f with domain $\mathbb{N} \times N \times D$, for some sets N and D.

In each session $i \in \{1, \ldots, n\}$, verifier V samples a value $n_i \in N$ uniformly at random and sends it to n_i. In each session, prover P then chooses (e.g., at

random) values $x_i \in D$ and checks whether $f(i, n_i, x_i)$ has at least d trailing zeroes; in that case, x_i is a valid solution for session i. Verifier V again checks the correctness and accepts only one attempt per session. We then prove the following theorem.

Theorem 2 (informal). *Let $d \in \mathbb{N}$ be the hardness parameter. If verified V can evaluate f at least n times, then the described protocol is a proof of effort with $\phi = 2^{-d}$, and where the bounds on prover attempts are as in the previous theorem.*

2.5 Non-interactive proofs of effort

In Section 8, we then continue to describing an analogous definition for a non-interactive proof of effort (niPoE), and again give an instantiation based on hash trail. The concept of a niPoEs stems from the work of Dwork and Naor [38], who proposed it as a measure of combatting junk mail. The idea is to associate a PoE, instead of to a particular interactive session as above, to a certain string that may consist of data such as, in the case of [38], the sender, receiver, and contents of the mail.

The non-interactive proof-of-effort resource. On a high level, the resource describing the goal of non-interactive proofs of effort is analogous to the one described above, with the main difference that the concept of a session is replaced by bit strings $s \in \{0,1\}^*$. The resource allows prover P to adaptively input up to a strings $s \in \{0,1\}^*$, with repetitions, and once the prover has allotted sufficient resources to a string s, the same string is output to the receiver. As for the interactive PoEs above, the prover can freely distribute its effort over different strings, but has an overall bounded number of attempts.

We again formalize these guarantees in a resource that can be accessed by P and V some number n of sessions in which they interact, and that is parametrized by a function $\phi : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and a number $a \in \mathbb{N}$. For each bit string $s \in \{0,1\}^*$, prover P can repeatedly attempt to convince verifier V; the probability for this to succeed in the i^{th} attempt within a session is $\phi(i)$, and P will learn whether the attempt was successful. Accumulated over all sessions, P has a attempts before the resource stops accepting further inputs. Verifier V can repeatedly query the resource for strings for which P's proof succeeded. For a dishonest prover, the interface provided by the resource is slightly different, analogously to the case of the interactive PoE. This intuitively described resource is made formal in Section 8.

We then prove that a protocol based on the hash-trail, which assumes the existence of a query-bounded random oracle and unidirectional communication from P to V, constructs the described niPoE resource.

The hash-trail scheme. The scheme is based on requiring the prover to provide an input to the MoHF that includes the target string s and leads to an output

of some form, such as some number $d \in \mathbb{N}$ of trailing zero bits [2]. We consider a MoHF f with domain $\{0,1\}^* \times D$, for some set D.

For each target string $s \in \{0,1\}^*$, prover P chooses (e.g., at random) values $x_i \in D$ and checks whether $f(s, x_i)$ has at least d trailing zeroes; in that case, x_i is a valid solution for target string s. Verifier V checks the correctness and accepts if the check verifies. We then prove the following theorem.

Theorem 3. *Let $d \in \mathbb{N}$ the hardness parameter. Then the described protocol is a non-interactive proof-of-effort with $\phi = 2^{-d}$.*

2.6 Combining the results

The main result of our paper then follows by composing the results on the MoHF constructions with the protocols for PoE and niPoE described above. Before we can compose the MoHFs with the application protocols using the respective composition theorem [62,61], however, we have to resolve one apparent incompatibility. The indifferentiability statement for MoHF is not immediately applicable in the case with two honest parties, as required in the correctness statements for PoE and niPoE where both the prover and verifier are honest. We further explain how to resolve this issue in Appendix A.

In more detail, for an $(\mathsf{a}, \mathsf{b}, \varepsilon)$-MoHF in some model, and a proof of effort parametrized by ϕ, the composition of the MoHF and the PoE construct the PoE resource described above with a attempts allowed to the prover P, and consequently $\alpha + n$ attempts for the dishonest prover and n sessions.

In summary, the results in this paper allow to combine different types of hardness with different types of applications; for instance, we immediately obtain PoEs for the settings of spam mail [38] and cryptographic puzzles [54], but based on memory-hardness [68], when instantiated with functions proposed in [7]. The greatest benefit of our approach is, however, that when using our framework, in the future any newly analyzed function can immediately be used in all previously analyzed applications, and any newly analyzed application can immediately use all previously analyzed functions.

3 Preliminaries for the full paper

We use the sets $\mathbb{N} := \{1, 2, \ldots\}$, and $\mathbb{Z}_{\geq c} := \{c, c+1, \ldots\} \cap \mathbb{Z}$ to denote integers greater than or equal to c. Similarly we write $[a, c]$ to denote $\{a, a+1, \ldots, c\}$ and $[c]$ for the set $[1, c]$. For a set S, we use the notation $x \leftarrow\!\!\$\, S$ to denote that x is chosen uniformly at random from the set S. For arbitrary set $\mathbb{I}$ and $n \in \mathbb{N}$ we write $\mathbb{I}^{\times n}$ to denote the n-wise cross product of $\mathbb{I}$. We refer to sets of functions (or distributions) as *function (or distribution) families*.

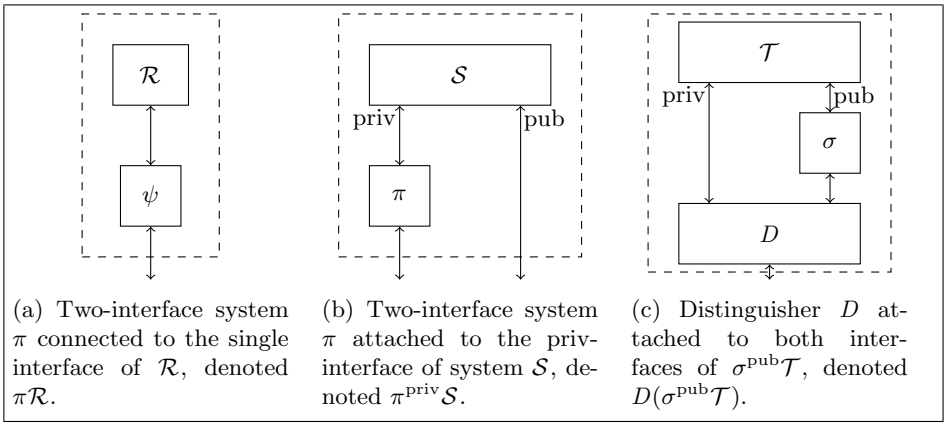
3.1 Reactive discrete systems

For an input set $\mathbb{X}$ and an output set $\mathbb{Y}$, a *reactive discrete* $(\mathbb{X}, \mathbb{Y})$*-system* repeatedly takes as input a value (or query) $x_i \in \mathbb{X}$ and responds with a value

$y_i \in \mathbb{Y}$, for $i \in \{1, 2, \dots\}$. Thereby, each output y_i may depend on all prior inputs $x_1, \dots, x_i$. As discussed by Maurer [60], reactive discrete systems are exactly modeled by the notion of a *random system*, that is, the conditional distribution $\mathsf{p}_{Y_i|X^iY^{i-1}}$ of each output (random variable) $Y_i \in \mathbb{Y}$ given all previous inputs $X_1, \dots, X_i \in \mathbb{X}$ and outputs $Y_1, \dots, Y_{i-1} \in \mathbb{Y}$ of the system.

Discrete reactive systems can have multiple interfaces, where each interface is labeled by an element in some set $\mathbb{I}$. We then formally consider $(\mathbb{I} \times \mathbb{X}, \mathbb{I} \times \mathbb{Y})$-systems, where providing an input $x \in \mathbb{X}$ at interface $i \in \mathbb{I}$ then means evaluating the system on input $(i, x) \in \mathbb{I} \times \mathbb{X}$, and the resulting output $(i', y) \in \mathbb{Y}$ means that the value y is provided as a response at the interface $i' \in \mathbb{I}$. We generally denote reactive discrete systems by upper-case calligraphic letters such as $\mathcal{S}$ or $\mathcal{T}$ or by lower-case Greek letters such as π or σ.

A *configuration of systems* is a set of systems which are connected via their interfaces. Any configuration of systems can again be seen as a system that provides all unconnected interfaces to its environment. Examples are shown in Figure 2, where Sub-figure 2a shows a two-interface system π connected to the single interface of another system $\mathcal{R}$, and Sub-figure 2b shows a two-interface system π connected to the priv-interface of the system $\mathcal{S}$. The latter configuration is denoted by the term $\pi^{\text{priv}}\mathcal{S}$. Finally, Sub-figure 2c shows a similar setting, but where additionally a distinguisher (or environment) D is attached to both interfaces of $\sigma^{\text{pub}}\mathcal{T}$. This setting is denoted as $D(\sigma^{\text{pub}}\mathcal{T})$ and is further discussed in Section 3.2.



(a) Two-interface system π connected to the single interface of $\mathcal{R}$, denoted $\pi\mathcal{R}$.

(b) Two-interface system π attached to the priv-interface of system $\mathcal{S}$, denoted $\pi^{\text{priv}}\mathcal{S}$.

(c) Distinguisher D attached to both interfaces of $\sigma^{\text{pub}}\mathcal{T}$, denoted $D(\sigma^{\text{pub}}\mathcal{T})$.

Fig. 2: Examples for configurations of systems.

3.2 Indifferentiability

The main definitions in this work are based on the indifferentiability framework of Maurer *et al.* [64,63]. We define the indifferentiability notion in this section;

for more information on the underlying systems model we refer to Appendix 3.1 or the literature [60,64].

Indifferentiability of a protocol or scheme π, which using certain resources $\mathcal{S}$, from resource $\mathcal{T}$ requires that there exists a simulator σ such that the two systems $\pi^{\text{pub}}\mathcal{S}$ and $\sigma^{\text{pub}}\mathcal{T}$ are indistinguishable. The indistinguishability is defined via a distinguisher D, a special system that interacts with either $\pi^{\text{priv}}\mathcal{S}$ or $\sigma^{\text{pub}}\mathcal{T}$ and finally outputs a bit. In the considered "real-world" setting with $\pi^{\text{priv}}\mathcal{S}$, the distinguisher D has direct access to the pub-interface of $\mathcal{S}$, but the priv-interface is accessible only through π. In the considered "ideal-world" setting with $\sigma^{\text{pub}}\mathcal{T}$, D has direct access to the priv-interface of $\mathcal{T}$, but the pub-interface is accessible only through σ. The advantage of the distinguisher is now defined to be the difference in the probability that D outputs some fixed value, say 1, in the two settings, more formally,

$$\Delta^D\left(\pi^{\text{priv}}\mathcal{S}, \sigma^{\text{pub}}\mathcal{T}\right) = \left|\Pr\left[D(\pi^{\text{priv}}\mathcal{S}) = 1\right] - \Pr\left[D(\sigma^{\text{pub}}\mathcal{T}) = 1\right]\right| \ .$$

Intuitively, if the advantage is small, then, for the honest parties, the real-world resource $\mathcal{S}$ is at least as useful (when using it via π) as the ideal-world resource $\mathcal{T}$. Conversely, for the adversary the real world is at most as useful as the ideal world. Put differently, from the perspective of the honest parties, the real world is at least as safe as the ideal world. So any application that makes use of $\mathcal{T}$ can instead use $\pi^{\text{priv}}\mathcal{S}$. This leads to the following definition.

Definition 3 (Indifferentiability). *Let π be a protocol and $\mathcal{S}, \mathcal{T}$ be resources, and let $\varepsilon > 0$. Then $\pi^{\text{priv}}\mathcal{S}$* is ε-indifferentiable from $\mathcal{T}$*, if*

$$\exists \sigma : \pi^{\text{priv}}\mathcal{S} \approx_{\varepsilon} \sigma^{\text{pub}}\mathcal{T} \ ,$$

with $\pi^{\text{priv}}\mathcal{S} \approx_{\varepsilon} \sigma^{\text{pub}}\mathcal{T}$ defined as $\forall D: \ \Delta^D\left(\pi^{\text{priv}}\mathcal{S}, \sigma^{\text{pub}}\mathcal{T}\right) \leq \varepsilon$.

3.3 Oracle functions and oracle algorithms

We explore several constructions of hard-to-compute functions that are defined via a sequence of calls to an oracle. To make this dependency explicit, we use the following notation. For sets D and R, a *random oracle (RO)* H is a random variable distributed uniformly over the function family $\mathbb{H} = \{h : D \to R\}$.

Definition 4 (Oracle functions). *For (implicit) oracle set $\mathbb{H}$, an* oracle function $f^{(\cdot)}$ (with domain D and range R)*, denoted $f^{(\cdot)} : D \to R$, is a set of functions indexed by oracles $h \in \mathbb{H}$ where each f^h maps $D \to R$.*

We fix a concrete function in the set $f^{(\cdot)}$ by fixing an oracle $h \in \mathbb{H}$ to obtain function $f^h : D \to R$. More generally, if $\boldsymbol{f} = (f_1^{(\cdot)}, \ldots, f_n^{(\cdot)})$ is an n-tuple of oracle functions then we write $\boldsymbol{f}^h$ to denote the n-tuple $(f_1^h, \ldots, f_n^h)$.

For an algorithm $\mathtt{A}$ we write $\mathtt{A}^h$ to make explicit that $\mathtt{A}$ has access to oracle h during its execution. We sometimes refer to algorithms that expect such access as *oracle algorithm*. We leave the precise model of computation for such algorithms unspecified for now as these will vary between concrete notions of MoHFs.

Example 1. The *prefixed hash chain of length* $c \in \mathbb{N}$ is an oracle function as

$$f^h_{\text{HC},c}: \quad D \to R, \quad x \mapsto h\Big(c \| h\big(c-1\| \dots h(1\|x)\dots\big)\Big) .$$

An algorithm $\mathtt{A}^{\text{HC}}$ that computes a hash chain of length c is described as initially evaluating h at the input $1\|x$, and then iteratively $(c-1)$ times on the outputs of the previous round, prefixing with the round index. ◊

3.4 Computation and computational cost

One main goal of this paper is to introduce a unifying definitional framework for MoHFs. For any concrete type of MoHF, we have to quantify the (real-world) resources required for performing computations such as evaluating the function.

Cost measures. For the remainder of this section, we let $(\mathcal{V}, 0, +, \leq)$ be a commutative group with a partial order $\leq$ such that the operation "+" is compatible with the partial order "$\leq$", meaning that $\forall a, b, c \in \mathcal{V} : a \leq b \Rightarrow a+c \leq b+c$. More concretely, we could consider $\mathcal{V} = \mathbb{Z}$ or $\mathcal{V} = \mathbb{R}$, but also $\mathcal{V} = \mathbb{R}^n$ for some $n \in \mathbb{N}$ if the computational cost cannot be quantified by a single value, for instance if we want to measure both the computational effort and the memory required to perform the task. We generally use the notation $\mathcal{V}_{\geq 0} \coloneqq \{v \in \mathcal{V} : 0 \leq v\}$.

The cost of computation. We later describe several MoHFs for differing notions of effort, where the hardness is defined using the following complexity notion based on a generic cost function. Intuitively a cost function assigns a non-negative real number as a cost to a given execution of an algorithm $\mathtt{A}$. More formally, let $\mathbb{A}$ be some set of algorithms (in some fixed computational model). Then an $\mathbb{A}$-*cost function* has the form $\mathsf{cost} : \mathbb{A} \times \{0,1\}^* \times \{0,1\}^* \to \mathcal{V}_{\geq 0}$. The first argument is an algorithm, the second fixes the input to the execution and the third fixes the random coins of the algorithm (and, in the ROM, also the random coins of the RO). Thus any such triple completely determines an execution which is then assigned a cost. Concrete examples include measuring the number of RO calls made by $\mathtt{A}$ during the execution, the number of cache misses during the computation [37,39] or the amount of memory (in bits) used to store intermediate values during the computation [11]. We write $y \overset{\mathsf{a}}{\twoheadleftarrow} \mathtt{A}(x; \$)$ if the algorithm $\mathtt{A}$ computes the output $y \in \{0,1\}^*$, when given input $x \in \{0,1\}^*$ and random coins $\$ \leftarrow_{\$} \{0,1\}^*$, with computation cost $\mathsf{a} \in \mathcal{V}$.

For concreteness we continue developing the example of a hash-chain of length c by defining an appropriate cost notion.

Example 2. Let $\mathtt{A}$ be an oracle algorithm as in Example 1. The cost of evaluating the algorithm $\mathtt{A}$ is measured by the number $b \in \mathbb{N} = \mathcal{V}$ of queries to the oracle that can be made during the evaluation of $\mathtt{A}$. Therefore, we write

$$y \overset{b}{\twoheadleftarrow}_{\#} \mathtt{A}^h(x)$$

if $\mathtt{A}$ computes y from x with b calls to the oracle h. For the algorithm $\mathtt{A}^{\mathrm{HC}}$ computing the prefixed hash chain of length $c \in \mathbb{N}$, the cost of each evaluation is c and therefore obviously independent of the choice of random oracle, so simply writing $y \overset{b}{\leftarrow\!\!\#}\ \mathtt{A}^{\mathrm{HC}}(x)$ is well-defined. $\Diamond$

3.5 A model for resource-bounded computation

In this section, we describe generically how we model resource-bounded computation in the remainder of this work. The scenario we consider in the following section has a party specify an algorithm and evaluate it, possibly repeatedly on different inputs. We want to model that evaluating the algorithm incurs a certain computational cost and that the party has bounded resources to evaluate the algorithm—depending on the available resources—only for a bounded number of times, or maybe not at all. Our approach consists of specifying a *computation device* to which an algorithm $\mathtt{A}$ can be input. Then, one can evaluate the algorithm repeatedly by providing inputs $x_1, \ldots, x_k$ to the device, which evaluates the algorithm $\mathtt{A}$ on each of the inputs. Each such evaluation incurs a certain computational cost, and as long as there are still resources available for computation, the device responds with the proper outputs $y_1 = \mathtt{A}(x_1), y_2 = \mathtt{A}(x_2), \ldots$. Once the resources are exhausted, the device always responds with the special symbol $\perp$. In the subsequent part of this paper, we will often refer to the computation device as the "computation resource."

The above-described approach can be used to model arbitrary types of algorithms and computational resources. Examples for such resources include the memory used during the computation (memory-hardness) or the number of computational steps incurred during the execution (computational hardness). Resources may also come in terms of "oracles" or "sub-routines" called by the algorithms, such as a random oracle, where we may want to quantify the number of queries to the oracle (query hardness).

As a concrete example, we describe the execution of an algorithm whose use of resources accumulates over subsequent executions:[10]

1. Let $b \in \mathcal{V}$ be the resources available to the party and $j = 1$.
2. Receive input $x_j \in \{0,1\}^*$ from the party.
3. Compute $y_j \overset{c}{\leftarrow} \mathtt{A}(x_j)$, for $c \in \mathcal{V}$. If $c \geq b$ then set $b \leftarrow 0$ and output $\perp$. Otherwise, set $b \leftarrow b - c$ and output y_j. Set $j \leftarrow j + 1$ and go to step 2.

We denote the resource that behaves as described above for the specific case of oracle algorithms that are allowed to make a bounded number $b \in \mathbb{N}$ of oracle queries by $\mathcal{S}_b^{\mathrm{OA}}$. For concreteness we show how to define an appropriate computational resource for reasoning about the hash-chain example.

[10] An example of this type of resource restriction is the cumulative number of oracle calls that the algorithm can make. Other resources may have different characteristics, such as a bound on the maximum amount of simultaneous memory use during the execution of the algorithm; which does not accumulate over multiple executions.

Example 3. We continue with the setting described in Examples 1 and 2, and consider the hash-chain algorithm $\mathtt{A}^{\mathrm{HC}}$ with a computational resource that is specified by the overall number $b \in \mathcal{V} = \mathbb{N}$ that can be made to the oracle.

In more detail, we consider the resource $\mathcal{S}_b^{\mathrm{OA}}$ described above. Upon startup, $\mathcal{S}_b^{\mathrm{OA}}$ samples a uniform $h \leftarrow\!\!\$\ \mathbb{H}$. Upon input of the oracle algorithm $\mathtt{A}$ (the type described in Example 1) into the computation resource, the party can query $x_1, x_2, \ldots$ and the algorithm $\mathtt{A}$ is evaluated, with access to h, on all inputs until b queries to h have been made, and subsequently only returns $\perp$.

For algorithm $\mathtt{A}^{\mathrm{HC}}$, chain length c, and resource $\mathcal{S}_b^{\mathrm{OA}}$ with $b \in \mathbb{N}$, the algorithm can be evaluated $\lfloor b/c \rfloor$ times before all queries are answered with $\perp$. $\Diamond$

4 Moderately hard functions

In this section, we combine the concepts introduced in Section 3 and state our definition of moderately hard function. The existing definitions of MoHF can be seen as formalizing that, with a given amount of resources, the function can only be evaluated a certain (related) number of times. Our definition is different in that it additionally captures that even an arbitrary computation with the same amount of resources cannot provide more (useful) results about the function than making the corresponding number of evaluations. This stronger statement is essential for proving the security of applications.

We base the definition of MoHFs on the notion of indifferentiability discussed in Section 3.2. In particular, the definition is based on the indistinguishability of a *real* and an *ideal* execution that we describe below. Satisfying such a definition will then indeed imply the desired statement, i.e., that the best the adversary can do is evaluate the function in the forward direction, and additionally that for each of these evaluations it must spend a certain amount of resources.

The resource is parametrized by bounds $\mathbf{l}, \mathbf{r} \in \mathbb{P}$. Initially, set $\mathtt{A}_{\mathrm{pub}} \leftarrow \perp$, $b \leftarrow 0$, $h \leftarrow\!\!\$\ \mathcal{H}$.

On input $\mathtt{A}$ at priv:
If $\mathtt{A}_{\mathrm{priv}} = \perp$ then
  $\mathtt{A}_{\mathrm{priv}} \leftarrow \mathtt{A}$

On input x at priv:
If $\mathtt{A}_{\mathrm{priv}} \neq \perp$ then
  Return $\perp$
$y \overset{c}{\leftarrow} \mathtt{A}_{\mathrm{priv}}^{h}(x)$
If $c > \mathbf{l}$ then $y \leftarrow \perp$
$\mathbf{l} \leftarrow \mathbf{l} - c$
Return y

On input $(\mathtt{A}, x)$ at pub:
If $\neg b$ then
  $b \leftarrow 1$
  $y \overset{c}{\leftarrow} \mathtt{A}^{h}(x)$
  If $c > \mathbf{r}$ then $y \leftarrow \perp$
  Return y

Fig. 3: Specification of the real-world resource $\mathcal{S}_{\mathbf{l},\mathbf{r}}$.

The *real-world resource* consists of resource-bounded computational devices that can be used to evaluate certain types of algorithms; one such resource at the priv- and one at the pub-interface. For such a resource $\mathcal{S}$ with bounds specified by $\mathbf{l}, \mathbf{r} \in \mathbb{P}$, for some parameter space $\mathbb{P}$ that is specified by $\mathcal{S}$, for the priv- and

pub-interfaces, respectively, we usually write $\mathcal{S}_{\mathbf{l},\mathbf{r}}$. The protocol system π used by the honest party initially inputs an algorithm $\mathtt{naïve}$ to $\mathcal{S}_{\mathbf{l},\mathbf{r}}$, further inputs $x_1, x_2, \ldots$ from D to π are simply forwarded to $\mathcal{S}_{\mathbf{l},\mathbf{r}}$, and the responses are given back to D. Moreover, D can use the pub-interface of $\mathcal{S}_{\mathbf{l},\mathbf{r}}$ to input an algorithm $\mathtt{A}'$ and evaluate it.

The resource is parametrized by bounds $a, b \in \mathbb{N}$. Initially, set $i, j \leftarrow 0$, and let $F : D \to R$ be empty.

On input $x \in D$ at priv:	On input $x \in D$ at pub:
If $i \geq a$ then return $\perp$	If $j \geq b$ then return $\perp$
$i \leftarrow i + 1$	$j \leftarrow j + 1$
If $F[x] \neq \perp$ then $F[x] \leftarrow\!\!\$\ R$	If $F[x] \neq \perp$ then $F[x] \leftarrow\!\!\$\ R$
Return $F[x]$	Return $F[x]$

Fig. 4: Lazy-sampling specification of the ideal-world resource $\mathcal{T}^{\text{RRO}}_{a,b}$.

The *ideal-world resource* also has two interfaces priv and pub. We consider only moderately hard functions with uniform outputs; therefore, the ideal-world resource $\mathcal{T}^{\text{RRO}}$ we consider essentially implements a random function $D \to R$ and allows at both interfaces simply to query the random function. (In more detail, $\mathcal{T}^{\text{RRO}}$ is defined as initially choosing a uniformly random function $f : D \to R$ and then, upon each input $x \in D$ at either priv or pub, respond with $f(x) \in R$ at the same interface.) We generally consider resources $\mathcal{T}^{\text{RRO}}_{a,b}$ for $a, b \in \mathbb{N}$, which is the same as a resource $\mathcal{T}^{\text{RRO}}$ allowing a queries at the priv and b queries at the pub-interface. All exceeding queries are answered with the special symbol $\perp$.

It is easy to see that the resource $\mathcal{T}^{\text{RRO}}_{a,b}$ is one-way: it is a random oracle to which a bounded number of queries can be made.

Before we provide a more detailed general definitions, we complete the hash-chain example by instantiating an appropriate security notion.

Example 4. We extend Example 3 where the algorithm $\mathtt{A}^{\text{HC}}$ evaluates a hash-chain of length c on its input by defining the natural security notion such an algorithm achieves. The real-world resource $\mathcal{S}^{2\text{OA}}_{a,b}$, with $a, b \in \mathbb{N}$, behaves as a resource $\mathcal{S}^{\text{OA}}_{a}$ at the priv- and as a resource $\mathcal{S}^{\text{OA}}_{b}$ at the pub-interface. That is $\mathcal{S}^{2\text{OA}}_{a,b}$ first samples a random function $h \in \mathbb{H}$ uniformly, and then uses this for the evaluation of algorithms input at both interfaces priv and pub analogously to $\mathcal{S}^{\text{OA}}_{a}$ and $\mathcal{S}^{\text{OA}}_{B}$, respectively.

The converter system π_{HC} initially inputs $\mathtt{A}^{\text{HC}}$ into $\mathcal{S}^{2\text{OA}}_{a,b}$; which is a resource that allows for evaluating such algorithms at both interfaces priv and pub. As $\mathcal{S}^{2\text{OA}}_{a,b}$ allows for a oracle queries for $\mathtt{A}^{\text{HC}}$, the system $\pi_{\text{HC}}{}^{\text{priv}}\mathcal{S}^{2\text{OA}}_{a,b}$ allows for $\lfloor a/c \rfloor$ complete evaluations of $\mathtt{A}^{\text{HC}}$ at the priv-interface. The resource $\mathcal{T}^{\text{RRO}}_{a',b'}$ is a random oracle that can be queried at both interfaces priv and pub (and indeed the outside interface provided by π is of that type). The simulator σ, therefore, will initially accept an algorithm $\mathtt{A}'$ as input and then evaluate $\mathtt{A}'$ with simulating

the queries to h potentially using queries to $\mathcal{T}^{\text{RRO}}_{a',b'}$. In particular, we can rephrase the statement about (prefixed) iteration of random oracles of Demay *et al.* [35] as follows[11]: with π_{HC} being the system that inputs the algorithm $\mathtt{A}^{\text{HC}}$, and $\mathcal{S}^{2\text{OA}}_{a,b}$ the resource that allows a and b evaluations of h at the priv- and pub-interfaces, respectively, ${\pi_{\text{HC}}}^{\text{priv}}\mathcal{S}^{2\text{OA}}_{a,b}$ is $(b \cdot 2^{-w})$-indifferentiable, where w is the output width of the oracle, from $\mathcal{T}^{\text{RRO}}_{a',b'}$ allowing $a' = \lfloor a/c \rfloor$ queries at the priv- and $b' = \lfloor b/c \rfloor$ queries at the pub-interface. $\Diamond$

The security statement ensures both that the honest party is able to perform its tasks using the prescribed algorithm and resource, and that the adversary *cannot* to perform *more* computations than allowed by its resources. We emphasize that the *ideal* execution in Example 4 will allow both the honest party and the adversary to query a random oracle for some bounded number of times. The fact that in the *real* execution the honest party can answer the queries with its bounded resource corresponds to the efficient implementation of the MoHF. The fact that any adversarial algorithm that has a certain amount of resources available can be "satisfied" with a bounded number of queries to the ideal random oracle means that the adversarial algorithm cannot gain more knowledge than by evaluating the ideal function for that number of times. Therefore, Example 4 models the basic properties that we require from a MoHF.

The security statement for an MoHF with naïve algorithm $\mathtt{naïve}$ has the following form. Intuitively, for resource limits $(\mathbf{l}, \mathbf{r})$, the real model with those limits and the ideal model with limits $(\mathsf{a}(\mathbf{l}), \mathsf{b}(\mathbf{r}))$ are ε-indistinguishable, for some $\varepsilon = \varepsilon(\mathbf{l}, \mathbf{r})$. I.e., there is a simulator σ such that no distinguisher D can tell the two models apart with advantage $> \varepsilon$.

We recall that the role of σ is to "fool" D into thinking it is interacting with A in the real model. We claim that this forces σ to be aware of the concrete parameters $\mathbf{r}$ of the real world D is supposedly interacting with. Indeed, one strategy D may employ is to provide code $\mathtt{A}$ at the pub-interface which consumes all available computational resources. In particular, using this technique D will obtain a view encoding $\mathbf{r}$. Thus it had better be that σ is able to produce a similar encoding itself. Thus in the following definition we allow σ to depend on the choice of $\mathbf{r}$. Conversely, no such dependency between $\mathbf{l}$ and σ is needed.[12]

For many applications, we also want to parametrize the function by a hardness parameter $\mathbf{n} \in \mathbb{N}$. In that case we consider a sequence of oracle functions $f^{(\cdot)}_{\mathbf{n}}$ and algorithms $\mathtt{naïve}_{\mathbf{n}}$ (which we will often want to be uniform) and also the functions $\mathsf{a}, \mathsf{b}, \varepsilon$ must be defined separately for each $\mathbf{n} \in \mathbb{N}$. This leads us to the following definition.

Definition 5 (MoHF security). *For each $\mathbf{n} \in \mathbb{N}$, let $f^{(\cdot)}_{\mathbf{n}}$ be an oracle function and $\mathtt{naïve}_{\mathbf{n}}$ be an algorithm for computing $f^{(\cdot)}$, let $\mathbb{P}$ be a parameter space and $\mathsf{a}, \mathsf{b} : \mathbb{P} \times \mathbb{N} \to \mathbb{N}$, and let $\varepsilon : \mathbb{P} \times \mathbb{P} \times \mathbb{N} \to \mathbb{R}_{\geq 0}$. Then, for a family of models*

[11] Similar statements have been proven earlier by Yao and Yin [82] and Bellare et al. [17]; however, we use the result on prefixed iteration from [35].

[12] We remark that in contrast to, say, non-black box simulators, we are unaware of any actual advantage of this independence between σ and $\mathbf{l}$.

$\mathcal{S}_{\mathbf{l},\mathbf{r}}$, $(f_{\mathbf{n}}^{(\cdot)}, \mathtt{naïve}_{\mathbf{n}})_{\mathbf{n}\in\mathbb{N}}$ *is a* $(\mathsf{a},\mathsf{b},\varepsilon)$-secure moderately hard function family in the $\mathcal{S}_{\mathbf{l},\mathbf{r}}$-model *if*

$$\forall \mathbf{n}\in\mathbb{N}, \mathbf{r}\in\mathbb{P}\ \exists\sigma\ \forall \mathbf{l}\in\mathbb{P}: \quad \pi^{\text{priv}}_{\mathtt{naïve}_{\mathbf{n}}}\ \mathcal{S}_{\mathbf{l},\mathbf{r}} \quad \approx_{\varepsilon(\mathbf{l},\mathbf{r},\mathbf{n})} \quad \sigma^{\text{pub}}\ \mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l},\mathbf{n}),\mathsf{b}(\mathbf{r},\mathbf{n})}\ ,$$

The function family is called uniform *if* $(\mathtt{naïve}_{\mathbf{n}})_{\mathbf{n}\in\mathbb{N}}$ *is a uniform algorithm. The function family is* asymptotically secure *if* $\varepsilon(\mathbf{l},\mathbf{r},\cdot)$ *is a negligible function in the third parameter for all values of* $\mathbf{r},\mathbf{l}\in\mathbb{P}$.

We sometimes use the definition with a fixed hardness parameter $\mathbf{n}$. Note also that the definition is fundamentally different from resource-restricted indifferentiability [34] in that there the simulator is restricted, as the idea is to *preserve* the same complexity (notion).

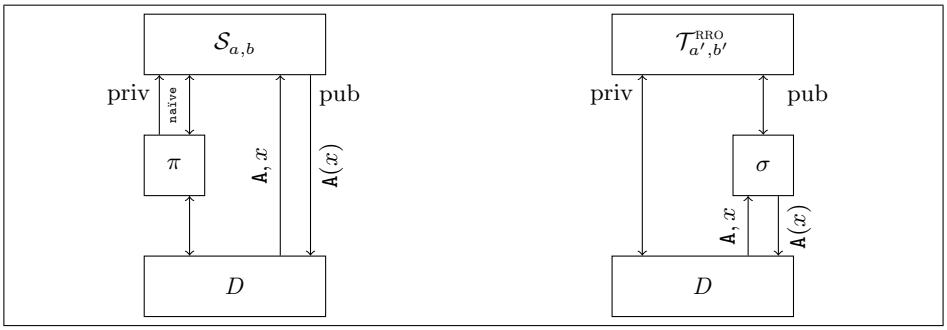


Fig. 5: Outline for the indifferentiability-based notion.

Further discussion on real model. In the real model, the resource described in Figure 3 is available to the (honest) party at the priv-interface and the adversarial party at the pub-interface. Since our goal is to model different types of computational hardness of specific tasks, that is, describe the amount of resources needed to perform these tasks, the nature of the remaining resources will naturally vary depending on the particular type of hardness being modeled. For example, when modeling memory-hardness, the computation resource would limit the amount of memory available during the evaluation, and a bound on the computational power available to the party would correspond to defining computational hardness. Each resource is parametrized by two values $\mathbf{l}$ and $\mathbf{r}$ (from some arbitrary parameter space $\mathbb{P}$) denoting limits on the amount of the resources available to the parties at the priv- and pub-interfaces, respectively.[13] Beyond the local computation resources described above, oracle algorithms have access to an oracle that is chosen initially in the resource according to the prescribed distribution and *the same* instance is made available to the algorithms

[13] These parameters may specify bounds in terms of the cost function discussed above.

at all interfaces. In this work, the algorithms will always have access to a random oracle, i.e. a resource that behaves like a random function h.

We generally denote the real-world resource by the letter $\mathcal{S}$ and use the superscript to further specify the type of computational resource and the subscript for the resource bounds, as $\mathcal{S}^{2\text{OA}}_{a,b}$ in Example 4, where $\mathbb{P} = \mathbb{N}$, $\mathbf{l} = a$ and $\mathbf{r} = b$.

Both interfaces priv and pub of the real-world resource expect as an input a program that will be executed using the resources specified at the respective interface. Suppose we wish to make a security statement about the hardness of a particular MoHF with the naïve algorithm `naïve`. Besides the resources themselves, the real world contains a system π that simply inputs `naïve` to be executed. Following the specification in Figure 3, the execution in the real model can be described as follows:

- Initially, D is activated and can evaluate `naïve` on inputs of its choice by providing inputs at the priv-interface.[14]
- Next, D can provide as input an algorithm `A` at the pub-interface, and evaluate `A` on one input x. The computation resource will evaluate `A` on input x.
- Next, D can again provide queries at the priv-interface to evaluate the algorithms `naïve` (until the resources are exhausted).
- Eventually, D outputs a bit (denoting its guess at whether it just interacted with the real world or not) and terminates.

At first sight, it might appear counter-intuitive that we allow the algorithm `A` input at pub to be evaluated only once, and not repeatedly, which would be stronger. The reason is that, for most complexity measures we are interested in, such as for memory-hard functions, continuous interaction with the environment D would allow `A` to "outsource" relevant resource-use to D, and contradict our goal of precisely measuring `A`'s resource consumption (and thereby sometimes render non-trivial statements impossible). This restriction can be relaxed wherever possible, as in Example 4.

Further discussion on ideal model. The (ideal-world) resource $\mathcal{T}$ also has a priv- and a pub-interface. In our definition of a MoHF, the ideal-world resource is always of the type $\mathcal{T}^{\text{RRO}}_{a,b}$ with $a, b \in \mathbb{N}$, that is, a random oracle that allows a queries at the priv- and b queries at the pub-interface. The priv-interface can be used by the distinguisher to query the oracle, while the pub-interface is accessed by the *simulator* system σ whose job it is to simulate the pub-interface of the real model consistently.

More precisely, for statements about parametrized real-world resources, we consider a class of ideal resources $\mathcal{T}^{\text{RRO}}_{\mathsf{a},\mathsf{b}}$ characterized by two functions a and b which map elements of $\mathbb{P}$ to $\mathbb{N}$. For any concrete real model given by parameters $(\mathbf{l}, \mathbf{r})$ we compare with the concrete ideal model with resource $\mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}),\mathsf{b}(\mathbf{r})}$

[14] Once the resources at the priv-interface are exhausted, no further useful information is gained by D in making additional evaluation calls for `naïve`.

parametrized by $(\mathsf{a}(\mathbf{l}), \mathsf{b}(\mathbf{r}))$. These numbers denote an upper bound on the number of queries to the random oracle permitted on the priv- and pub-interfaces, respectively. In particular, after $\mathsf{a}(\mathbf{l})$ queries on the priv-interface all future queries on that interface are responded to with $\perp$ (and similarly for the pub-interface with the limit $\mathsf{b}(\mathbf{r})$).

To a distinguisher D, an execution with the ideal model looks as follows:

- Initially, D is activated, and can make queries to $\mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}),\mathsf{b}(\mathbf{r})}$ at the priv-interface. (After $\mathsf{a}(\mathbf{l})$ queries $\mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}),\mathsf{b}(\mathbf{r})}$ always responds with $\perp$.)
- Next, D can provide as input an algorithm $\mathtt{A}$ at the pub-interface. Overall, the simulator σ can make at most $\mathsf{b}(\mathbf{r})$ queries to $\mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}),\mathsf{b}(\mathbf{r})}$.
- Next, D can make further queries to $\mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}),\mathsf{b}(\mathbf{r})}$ on the priv-interface.
- Finally, D outputs a bit (denoting its guess at whether it just interacted with the real world or not) and terminates.

An ideal model is outlined in Figure 5 with priv and pub resource limits a' and b' respectively.

5 Memory-hard functions

Moving beyond the straightforward example of an MoHF based on computational hardness developed during the above examples, we describe more advanced types of MoHFs in this and the next section. Each one is based on a different complexity notion and computational model. For each one, we describe one (or more) constructions. Moreover, for the first two we provide a powerful tool for constructing provably secure MoHFs of those types. We begin, in this section, with memory-hard functions (MHF).

In the introduction, we discussed shortcomings of the existing definitions of MHFs. We address these concerns by insantiating MHFs within our general MoHF framework and providing a pebbling reduction with which we can "rescue" the MHF constructions [11,7,6] and security proofs [7,6] of several recent MHFs from the literature. More generally, the tool is likely to prove useful in the future as new, more practical graphs are developed [6] and/or new labeling functions are developed beyond an ideal compression function. (For more details what is meant by "rescue" we refer to discussion immediately after Theorem 4.)

5.1 The parallel ROM

To define an MHF, we consider a resource-bounded computational device $\mathcal{S}$ with a priv- and a pub-interface capturing the pROM (adapted from [9]). Let $w \in \mathbb{N}$. Upon startup, $\mathcal{S}^{w\text{-PROM}}$ samples a fresh random oracle $h \leftarrow\!\!\$\ \mathbb{H}_w$ with range $\{0,1\}^w$. Now, on both interfaces, $\mathcal{S}^{w\text{-PROM}}$ accepts as input a pROM algorithm $\mathtt{A}$ which is an oracle algorithm with the following behavior.

A *state* is a pair $(\tau, \mathbf{s})$ where *data* τ is a string and $\mathbf{s}$ is a tuple of strings. The output of step i of algorithm $\mathtt{A}$ is an *output state* $\bar{\sigma}_i = (\tau_i, \mathbf{q}_i)$ where $\mathbf{q}_i =$

$[q_i^1, \ldots, q_i^{z_i}]$ is a tuple of *queries* to h. As input to step $i+1$, algorithm $\mathtt{A}$ is given the corresponding *input state* $\sigma_i = (\tau_i, h(\mathbf{q}_i))$, where $h(\mathbf{q}_i) = [h(q_i^1), \ldots, h(q_i^{z_i})]$ is the tuple of *responses* from h to the queries $\mathbf{q}_i$. In particular, for a given h and random coins of $\mathtt{A}$, the input state σ_{i+1} is a function of the input state σ_i. The initial state σ_0 is empty and the input x_{in} to the computation is given a special input in step 1.

For a given execution of a pROM, we are interested in the following complexity measure. We denote the bit-length of a string s by $|s|$. The *length* of a state $\sigma = (\tau, \mathbf{s})$ with $\mathbf{s} = (s^1, s^2, \ldots, s^y)$ is $|\sigma| = |\tau| + \sum_{i \in [y]} |s^i|$. The *cumulative memory complexity* (CMC) of an execution is the sum of the lengths of the states in the execution. More precisely, let us consider an execution of algorithm $\mathtt{A}$ on input x_{in} using coins \$ with oracle h resulting in $z \in \mathbb{Z}_{\geq 0}$ input states $\sigma_1, \ldots, \sigma_z$, where $\sigma_i = (\tau_i, \mathbf{s}_i)$ and $\mathbf{s}_i = (s_i^1, s_i^2, \ldots, s_i^{y_j})$. Then the *cumulative memory complexity* (CMC) of the execution is

$$\mathsf{cmc}(\mathtt{A}^h(x_{\mathsf{in}}; \$)) = \sum_{i \in [z]} |\sigma_i| \ ,$$

while the *total number of RO calls* is $\sum_{i \in [z]} y_j$. More generally, the CMC (and total number of RO calls) of several executions is the sum of the CMC (and total RO calls) of the individual executions.

We now describe the resource constraints imposed by $\mathcal{S}^{w\text{-PROM}}$ on the pROM algorithms it executes. To quantify the constraints, $\mathcal{S}^{w\text{-PROM}}$ is parametrized by a left and a right tuple from the following parameter space $\mathbb{P}^{\text{PROM}} = (\mathbb{Z}_{\geq 0})^2$ describing the constraints for the priv and pub interfaces respectively. In particular, for parameters $(q, m) \in \mathbb{P}^{\text{PROM}}$, the corresponding pROM algorithm is allowed to make a total of q RO calls and use CMC at most m summed up across all of the algorithms executions.[15]

As usual for memory-hard functions, to ensure that the honest algorithm can be run on realistic devices, $\mathcal{S}^{w\text{-PROM}}$ restricts the algorithms on the priv-interface to be *sequential*. That is, the algorithms can make only a single call to h per step. Technically, in any execution, for any step j it must be that $y_j \leq 1$. No such restriction is placed on the adversarial algorithm reflecting the power (potentially) available to such a highly parallel device as an ASIC.

We conclude the section with the formal definition of a memory-hard function in the pROM. The definition is a particular instance of an MoHF defined in Definition 5 formulated in terms of exact security.

Definition 6 ((Parallel) memory-hard function). *For each $\mathbf{n} \in \mathbb{N}$, let $f_{\mathbf{n}}^{(\cdot)}$ be an oracle function and $\mathtt{naïve}_{\mathbf{n}}$ be a pROM algorithm for computing $f^{(\cdot)}$. Consider the function families:*

$$\mathsf{a} = \{\mathsf{a}_w : \mathbb{P}^{\text{PROM}} \times \mathbb{N} \to \mathbb{N}\}_{w \in \mathbb{N}} \ , \quad \mathsf{b} = \{\mathsf{b}_w : \mathbb{P}^{\text{PROM}} \times \mathbb{N} \to \mathbb{N}\}_{w \in \mathbb{N}} \ ,$$

[15] In particular, for the algorithm input on the adversarial interface pub the single permitted execution can consume at most $\mathbf{r}$ resources while for the honest algorithm input on priv the total consumed resources across all execution can be at most $\mathbf{l}$.

$$\epsilon = \{\epsilon_w : \mathbb{P}^{\text{PROM}} \times \mathbb{P}^{\text{PROM}} \times \mathbb{N} \to \mathbb{R}_{\geq 0}\}_{w \in \mathbb{N}} .$$

Then $F = (f_{\mathbf{n}}^{(\cdot)}, \mathtt{naïve}_{\mathbf{n}})_{\mathbf{n} \in \mathbb{N}}$ *is called an* $(\mathsf{a}, \mathsf{b}, \epsilon)$-memory-hard function *(MHF) if* $\forall w \in \mathbb{N}$ F *is an* $(\mathsf{a}_w, \mathsf{b}_w, \epsilon_w)$*-secure moderately hard function family for* $\mathcal{S}^{w\text{-PROM}}$.

Data-(in)dependent MHFs. An important distinction in the literature of memory-hard functions concerns the memory-access pattern of `naïve`. In particular, if the pattern is independent of the input x then we call this a *data-independent* MHF (iMHF) and otherwise we call it an *data-dependent* MHF (dMHF). The advantage of an iMHF is that the honest party running `naïve` is inherently more resistant to certain side-channel attacks (such as cache-timing attacks) which can lead to information leakage about the input x. When the MHF is used for, say, password hashing on a login server this can be a significant concern. Above, we have chosen to not make the addressing mechanism used to store a state σ explicit in $\mathcal{S}^{w\text{-PROM}}$, as it would significantly complicate the exposition with little benefit. Yet, we remark that doing so would definitely be possible within the wider MoHF framework presented here if needed. Moreover the tools for constructing MHFs below actually construct iMHFs.

5.2 Graph functions

Now that we have a concrete definition in mind, we turn to constructions. We first define a large class of oracle functions (called graph functions) which have appeared in various guises in the literature [39,42,11] (although we differ slightly in some details which simplify later proofs). This allows us to prove the main result of this section; namely a "pebbling reduction" for graph functions. That is, for a graph function F based on some graph G, we show function families $(\mathsf{a}, \mathsf{b}, \epsilon)$ depending on G, for which function F is an MHF.

We start by formalizing (a slight refinement of) the usual notion of a graph function (as it appears in, say, [42,11]). For this, we use the following common notation and terminology. For a directed acyclic graph (DAG) $G = (V, E)$, we call a node with no incoming edges a *source* and a node with no outgoing edges a *sink*. The *in-degree* of a node is the number of its incoming edges and the in-degree of G is the maximum in-degree of any of its nodes. The *parents* of a node v are the set of nodes with outgoing edges leading to v. We also implicitly associate the elements of V with unique strings.[16]

A graph function makes use of a oracle $h \in \mathbb{H}_w$ defined over bit strings. Technically, we assume an implicit prefix-free encoding such that h is evaluated on unique strings. Inputs to h are given as distinct tuples of strings (or even tuples of tuples of strings). For example, we assume that $h(0, 00)$, $h(00, 0)$, and $h((0, 0), 0)$ all denote distinct inputs to h.

[16] For example, we can associate $v \in V$ with the binary representation of its position in an arbitrary fixed topological ordering of G.

Definition 7 (Graph function). *Let function* $h : \{0,1\}^* \to \{0,1\}^w \in \mathbb{H}_w$ *and DAG* $G = (V, E)$ *have source nodes* $\{v_1^{\mathsf{in}}, \ldots, v_a^{\mathsf{in}}\}$ *and sink nodes* $(v_1^{\mathsf{out}}, \ldots, v_z^{\mathsf{out}})$. *Then, for inputs* $\mathbf{x} = (x_1, \ldots, x_a) \in (\{0,1\}^*)^{\times a}$, *the* $(h, \mathbf{x})$-labeling *of* G *is a mapping* $\mathsf{lab} : V \to \{0,1\}^w$ *defined recursively to be:*

$$\forall v \in V \quad \mathsf{lab}(v) := \begin{cases} h(\mathbf{x}, v, x_j)) & : v = v_j^{\mathsf{in}} \\ h(\mathbf{x}, v, \mathsf{lab}(v_1), \ldots, \mathsf{lab}(v_d))) & : \textit{else} \end{cases}$$

where $\{v_1, \ldots, v_d\}$ *are the parents of* v *arranged in lexicographic order.*

The graph function (of G and $\mathbb{H}_w$) *is the oracle function*

$$f_G : (\{0,1\}^*)^{\times a} \to (\{0,1\}^w)^{\times z} ,$$

which maps $\mathbf{x} \mapsto (\mathsf{lab}(v_1^{\mathsf{out}}), \ldots, \mathsf{lab}(v_z^{\mathsf{out}}))$ *where* lab *is the* $(h, \mathbf{x})$*-labeling of* G.

The above definition differs from the one in [11] in two ways. First, it considers graphs with multiple source and sink nodes. Second it prefixes all calls to h with the input $\mathbf{x}$. This ensures that, given any pair of distinct inputs $\mathbf{x}_1 \neq \mathbf{x}_2$, no call to h made by $f_G(\mathbf{x}_1)$ is repeated by $f_G(\mathbf{x}_2)$. Intuitively, this ensures that finding collisions in h can no longer help avoiding making a call to h for each new label being computed. Technically, it simplifies proofs as we no longer need to compute and carry along the probability of such a collision. We remark that this is merely a technicality and if, as done in practice, the prefixing (of both $\mathbf{x}$ and the node v) is omitted, security will only degrade by a negligible amount.[17]

The naïve algorithm. The naïve oracle algorithm $\mathtt{naïve}_G$ for f_G computes one label of G at a time in topological order appending the result to its state. If G has $|V| = n$ nodes then $\mathtt{naïve}_G$ will terminate in n steps making at 1 call to h per step, for a total of n calls, and will never store more than $w(i-1)$ bits in the data portion of its state in the ith round. In particular for all inputs $\mathbf{x}$, oracles h (and coins \$) we have that $\mathsf{cmc}(\mathtt{naïve}_G^h(\mathbf{x}; \$)) = wn(n-1)/2$. Therefore, in the definition of an MHF we can set $\mathsf{a}_w(q, m) = \min(\lfloor q/n \rfloor, \lfloor 2m/wn(n-1) \rfloor)$. It remains to determine how to set b_w and ϵ_w, which is the focus of the next section.

5.3 A parallel memory-hard MoHF

In this section, we prove a pebbling reduction for memory hardness of a graph function f_G in the pROM. To this end, we first recall the parallel pebbling game over DAGs and associated cumulative pebbling complexity (CPC).

[17] Prefixing ensures domain separation; that is random oracle calls in a labeling are unique to that input. However, if inputs are chosen independently of the RO then finding two inputs that share an oracle call requires finding a collision in the RO. To concentrate on the more fundamental and novel aspects of the proofs below, we have chosen to instead assume full prefixing. A formal analysis with less prefixing can be found in [11].

The parallel pebbling game. The sequential version of the following pebbling game first appeared in [50,31] and the parallel version in [11]. Put simply, the game is a variant of the standard black-pebbling game where pebbles can be placed according to the usual rules but in batches of moves performed in parallel rather than one at a time sequentially.

Definition 8 (Pebbling a graph). *Let $G = (V, E)$ be a DAG and $T, S \subseteq V$ be node sets. Then a* (legal) pebbling of G (with starting configuration S and target T) *is a sequence $P = (P_0, \ldots, P_t)$ of subsets of V such that:*

1. *$P_0 \subseteq S$.*
2. *Pebbles are added only when their predecessors already have a pebble at the end of the previous step.*

$$\forall i \in [t] \quad \forall (x, y) \in E \quad \forall y \in P_i \setminus P_{i-1} \qquad x \in P_{i-1} \ .$$

3. *At some point every target node is pebbled (though not necessarily simultaneously).*

$$\forall x \in T \quad \exists z \leq t \qquad x \in P_z \ .$$

We call a pebbling of G complete *if $S = \emptyset$ and T is the set of sink nodes of G. We call a pebbling* sequential *if no more than one new pebble is placed per step,*

$$\forall i \in [t] \quad |P_i \setminus P_{i-1}| \leq 1 \ .$$

In this simple model of computation we are interested in the following complexity notion for DAGs taken from [11].

Definition 9 (Cumulative pebbling complexity). *Let G be a DAG, $P = (P_0, \ldots, P_t)$ be an arbitrary pebbling of G, and Π be the set of all complete pebblings of G. Then the* (pebbling) cost *of P and the* cumulative pebbling complexity *(CPC) of G are defined respectively to be:*

$$\mathsf{cpc}(P) := \sum_{i=0}^{t} |P_i| \ , \qquad \mathsf{cpc}(G) := \min \{\mathsf{cpc}(P) : P \in \Pi\} \ .$$

A pebbling reduction for memory-hard functions. We are now ready to formally state and prove the main technical result: a security statement showing a graph function to be an MHF for parameters $(\mathsf{a}, \mathsf{b}, \epsilon)$ expressed in terms of the CPC of the graph and the number of bits in the output of h.

Theorem 4 (Pebbling reduction). *Let $G_n = (V_n, E_n)$ be a DAG of size $|V_n| = n$. Let $F = (f_{G,n}, \mathtt{naïve}_{G,n})_{n \in \mathbb{N}}$ be the graph functions for G_n and their naïve oracle algorithms. Then, for any $\lambda \geq 0$, F is an $(\mathsf{a}, \mathsf{b}, \epsilon)$-memory-hard function where*

$$\mathsf{a} = \{\mathsf{a}_w(q, m) = \min(\lfloor q/n \rfloor, \lfloor 2m/wn(n-1) \rfloor)\}_{w \in \mathbb{N}} \ ,$$

$$\mathsf{b} = \left\{\mathsf{b}_w(q, m) = \frac{m(1+\lambda)}{\mathsf{cpc}(G)(w - \log q)}\right\}_{w \in \mathbb{N}} \ , \quad \epsilon = \left\{\epsilon_w(q, m) \leq \frac{q}{2^w} + 2^{-\lambda}\right\}_{w \in \mathbb{N}} \ .$$

We note that cpc charges for keeping pebbles on G which, intuitively, models storing the label of a node in the data component of an input state. However the complexity notion cmc for the pROM also charges for the responses to RO queries included in input states. We discuss three options to address this discrepancy.

1. Modify our definition of the pROM to that used in [11]. There, the i^{th} batch of queries $\mathbf{q}_i$ to h is made *during* step i. So the state stored between steps only contains the data component τ_i. Thus cmc in that model is more closely modeled by cpc. While the techniques used below to prove Theorem 4 carry over essentially unchanged to that model, we have opted to not go with that approach as we believe the version of the pROM used here (and in [8]) more closely captures computation for an ASIC. That is, it better models the constraint that during an evaluation of the hash function(s) a circuit must store any remaining state it intends to make use of later in separate registers. Moreover, given the depth of the circuit of hash functions used to realize h, at least one register per output bit of h will be needed.[18]
2. Modify the notion of cpc to obtain cpc', which also charges for new pebbles being placed on the graph. That is use $\mathsf{cpc}' = \mathsf{cpc} + \sum_i |P_i \setminus P_{i-1}|$ as the pebbling cost.[19] Such a notion would more closely reflect the way cmc is defined in this work. In particular, it would allow for a tighter lower bound in Theorem 4, since for any graph $\mathsf{cpc}' \geq \mathsf{cpc}$. Moreover, it would be easy to adapt the proof of Theorem 4 to accommodate cpc'. Indeed, (using the terminology from the proof of Theorem 4) in the ex-post-facto pebbling P of an execution, a node $v \notin P^x_{i-1}$ is only added to P^x_i if it becomes necessary for x at time i. By definition, this can only happen if there is a correct call for (x, v) in the input state σ_i. Thus, we are guaranteed that for each time step i it holds that $\sum_i \sum_x |P^x_i \setminus P^x_{i-1}| \leq y_i$, where y_i is the number of queries to h in input state σ_i. So we can indeed modify the second claim in the proof to also add the quantity $\sum_x |P^x_i \setminus P^x_{i-1}|$ to the left side of the inequality. The downside of this approach is that using cpc' in Theorem 4 would mean that it is no longer (immediately) clear if we can use any past results from the literature about cpc.
3. The third option, which we have opted for in this work, is to borrow from the more intuitive formulation of the pROM of [8] while sticking with the traditional pebbling complexity notion of cpc. We do this because, on the one hand, for any graph $\mathsf{cpc}' \leq 2\mathsf{cpc}$, so at most a factor of 2 is lost the tightness of Theorem 4 when using cpc instead of cpc'. Yet on the other hand, for cpc we already have constructions of graphs with asymptotically maximal cpc as well as a variety of techniques for analyzing the cpc of graphs. In particular we have upper and lower bounds for the cpc of arbitrary DAGs as well as for many specific graphs (and graph distributions) used in the literature as the

[18] Note that any signal entering a circuit at the beginning of a clock cycle that does not reach a memory cell before the end of a clock cycle is lost. Yet, hash functions so complex and clock cycles so short that it is unrealistic to assume an entire evaluation of h can be performed within a single cycle.

[19] cpc' is essentially the special case of "energy complexity" for $R = 1$ in [4].

basis for interesting graph functions [11,4,10,5,7]. Thus we have opted for this route so as to (A) strengthen the intuition underpinning the model of computation, (B) leave it clear that Theorem 4 can be used in conjunction with all of the past concerning cpc while (C) only paying a small price in the tightness of the bound we show in that theorem.

The remainder of this subsection is dedicated to proving the theorem. For simplicity we will restrict ourselves to DAGs with a single source $v_{\in}$ and sink v_{out} but this only simplifies notation. The more general case for any DAG is identical. The rough outline of the proof is as follows. We begin by describing a simulator σ as in Definition 5, whose goal is to simulate the pub-interface of $\mathcal{S}^{w\text{-PROM}}$ to a distinguisher D while actually being connected to the pub-interface of $\mathcal{T}^{\text{RRO}}$. In a nutshell, σ will emulate the algorithm $\mathtt{A}$ it is given by D internally by emulating a copy of $\mathcal{S}^{w\text{-PROM}}$ to it. σ will keep track of the RO calls made by $\mathtt{A}$ and, whenever $\mathtt{A}$ has made all the calls corresponding to a complete and legal (x, h)-labeling of G, then σ will query $\mathcal{T}^{\text{RRO}}$ at point x and return the result to $\mathtt{A}$ as the result of the final RO call for that labeling.

To prove that σ achieves this goal (with high probability) we introduce a generalization of the pebbling game, called an m-color pebbling, and state a trivial lemma showing that the cumulative m-color pebbling complexity of a graph is m times the CC of the graph. Next, we define a mapping between a sequence of RO calls made during an execution in the pROM (such as that of $\mathtt{A}$ being emulated by σ) and an m-coloring P of G. We prove a lemma stating that, w.h.p., if m distinct I/O pairs for f_G were produced during the execution, then P is legal and complete. We also prove a lemma upper-bounding the pebbling cost of P in terms of the CMC (and number of calls made to the RO) of the execution. But since the pebbling cost of G cannot be smaller than $m \cdot \mathsf{cpc}(G)$, this gives us a lower bound on the memory cost of any such execution, as desired. Indeed, any algorithm in the pROM that violates our bound on memory cost with too high probability implies the existence of a pebbling of G with too low pebbling cost, contradicting the pebbling complexity of G. But this means that when σ limits CMC (and number of RO calls) of the emulation of $\mathtt{A}$ accordingly, then w.h.p. we can upper-bound the number of calls σ will need to to $\mathcal{T}^{\text{RRO}}$.

To complete the proof, we have to show that using the above statements about σ imply that indifferentiability holds. Indeed, the simulation, conditioned on the events that no lucky queries occur and that the simulator does not need excessive queries, is perfect. Therefore, the distinguishing advantage can be bounded by the probability of provoking either of those events, which can be done by the above statements about σ. A detailed proof appears in Appendix B.

6 Other types of MoHFs

Besides MHFs, several other types of MoHFs have been considered in the literature. In this section, we briefly review weak memory-hard functions and memory-bound functions. A discussion of one-time computable functions and uncomputable functions is given in Appendix 6.3.

6.1 Weak memory-hard functions

A class of MoHFs considered in the literature that are closely related to MHFs are *weak* MHFs. Intuitively, they differ from regular MHFs only in that they also restrict adversaries to being sequential.[20] On the one hand, it may be easier to construct such functions compared to full blown MHF. In fact, for the data-independent variant of MHFs, [4] proves that a graph function based on a DAG of size n always has cmc of $O(wn^2/\log(n))$ (ignoring $\log\log$ factors). Yet, as discussed below, the results of [56,44] and those described below show that we can build W-MHFs from similar DAGs with sequential cmc of $\mathcal{O}(2n^2)$. Put differently, W-MHFs allow for strictly more memory consumption per call to the RO than is possible with MHFs. This is valuable since the limiting factor for an adversary is often the memory consumption while the cost for honest parties to enforce high memory consumption is the number of calls they must perform to the RO.

Weak MHFs as moderately hard functions. We capture weak MHFs in the MoHF framework as follows. The real world resource-bounded computational device $\mathcal{S}^{w\text{-SROM}}$ modeling the *sequential random oracle model* (sROM) is identical to $\mathcal{S}^{w\text{-PROM}}$ except that the adversarial algorithm it gets as input on the pub-interface must be sequential (just like the honest algorithm). In the notation used above to define the pROM in Section 5.1, in any execution of $\mathtt{A}$ during any step j, the output state must be such that $y_j \leq 1$. The parameter space is $\mathbb{P}^{\text{SROM}} = \mathbb{P}^{\text{PROM}}$. We refer to MHFs that satisfy the Definition 6 for $\mathcal{S}^{w\text{-SROM}}$ as *weak memory-hard functions* W-MHFs.

Given this definition we can now easily adapt the pebbling reduction of Theorem 4 to obtain a tool for constructing W-MHFs. In particular let scpc be the same as cpc except that in Definition 9 we set Π be the set of all *sequential* pebblings of G. The proof of Theorem 4 applies (essentially unchanged) to the new pebbling complexity notion; the only aspect we have to check is that if $\mathtt{A}$ is an sROM algorithm, then the ex-post-facto pebbling of any execution of $\mathtt{A}$ is always a sequential m-pebbling. But this holds because a pebble is only added to G whenever a new node becomes necessary, which is only the case when a correct call for that node is made to the RO. Since $\mathtt{A}$ can only make one call per step, at most one new node can become necessary.

Applications of the pebbling reduction. The pebbling reduction W-MHFs has some immediate implications. In [56], Lengaur and Tarjan prove that the DAGs underlying the two graph functions Catena Dragonfly and Butterfly [44] have $\mathsf{scpc} = \mathcal{O}(n^2)$. In [44], the authors extend these results to analyze the scpc of stacks of these DAGs. By combining those results with the pebbling reduction for the sROM, we obtain parameters $(\mathsf{a}, \mathsf{b}, \epsilon)$ for which the Catena functions are provably W-MHFs. Similar implications hold for the pebbling analysis done for the Balloon Hashing function in [23].

[20] If the adversary is restricted to using general-purpose CPUs and not ASICs or FPGAs with their massive parallelism, this restriction may be reasonable.

6.2 Memory-bound functions

Another important notion of MoHF from the literature has been considered in [37,39]. These predate MHFs and are based on the observation that while computation speeds vary greatly across real-world computational devices, this is much less so for memory-access speeds. Under the assumption that time spent on a computation correlates with the monetary cost of the computation, this observation motivates measuring the cost of a given execution by the number of cache misses (i.e., memory accesses) made during the computation. A function that requires a large number of misses, regardless of the algorithm used to evaluate the function, is called a *memory-bound* function.

Memory-bound functions as MoHFs. We show how to formalize memory-bound functions in the MoHF framework. In particular, we describe the real-world resource-bounded computational device $\mathcal{S}^{w\text{-MB}}$. It makes use of RO with w-bits of output and is parametrized by 6 positive integers $\mathbb{P}^{\text{MB}} = \mathbb{N}^{\times 6}$. That is, following the model of [39], an algorithm $\mathtt{A}$, executed by $\mathcal{S}^{w\text{-MB}}$ with parameters (m, b, s, ω, c, q), makes a sequence of calls to the RO and has access to a two tiered memory consisting of a cache of limited size and a working memory (as large as needed). The memory is partitioned into m blocks of b bits each, while cache is divided into s words of ω bits each. When $\mathtt{A}$ requests a location in memory, if the location is already contained in cache, then $\mathtt{A}$ is given the value for free, otherwise the block of memory containing that location is fetched into cache. The algorithm is permitted a total of q calls to the RO and c fetches (i.e. cache misses) across all executions.

In [37,39] the authors describe such functions (with several parameters each) and prove that the hash-trail construction applied to these functions results in a PoE for a notion of "effort" captured by memory-boundedness. (See Section 7 for more on the hash-trail construction and PoEs). We conjecture that the proofs in those works carry over to the notion of memory-bound MoHFs described above (using some of the techniques at the end of the proof of Theorem 4). Yet, we believe that a more general pebbling reduction (similar to Theorem 4) is possible for the above definition. Such a theorem would allow us to construct new and improved memory-bound functions. (On the one hand, the function described in [37] has a large description—many megabytes—while the function in [39] is based on superconcentrators which can be somewhat difficult to implement in practice with optimal constants.) In any case, we believe investigating memory-bound functions as MoHFs to be an interesting and tractable line of future work.

6.3 One-time computable and uncomputable functions

Another—less widely used—notion of MoHFs appearing in the literature are *one-time computable* functions [42]. Intuitively, these are sets of T pseudo-random functions (PRFs) $f_1, \ldots, f_T$ with long keys (where T is an *a priori* fixed, arbitrary number). An honest party can evaluate each function f_i exactly once, using

a device with limited memory containing these keys. On such a device, evaluating the i^{th} PRF provably requires deleting all of the first i keys. Therefore, if an adversary (with arbitrary memory and computational power) can only learn a limited amount of information about the internal state of the device, then regardless of the computation performed on the device, the adversary will never learn more than one input/output pair per PRF. The authors describe the intuitive application of a password-storage device secure against dictionary attacks. An advantage of using the MoHF framework to capture one-time computable functions could be proving security for such an application (using the framework's composition theorem).

We adapt the computational model of [42] to capture one-time computable function as MoHFs. We also generalize it to allow parties access to multiple memory-constrained devices. Then using the NIPOE construction below we describe a new (hypothetical) application for one-time computable functions to anonymous digital payment systems.

The real-world $\mathcal{S}^{(w,\,n)\text{-OC}}$ accepts algorithms of the type $\mathtt{A} = (\mathtt{A}_{\text{big}}, \{\mathtt{A}_i\}_{i\in[n]})$. Algorithm $\mathtt{A}_i$ is run on a sub-device containing the (freshly sampled) keys for PRFs $f_{i,j}$ with $j \in [z_i]$. The sub-device permits access to a RO h but is constrained to having s_i bits of memory. Algorithm $\mathtt{A}_{\text{big}}$ runs on a sub-device with access to h and no constraint on its memory. The input to the computation has the form $x_{\mathsf{in}} = (\mathbf{x}_1, \ldots, \mathbf{x}_z)$ where $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,i_z})$. Algorithm $\mathtt{A}_i$ is run on input $\mathbf{x}_i$ and can exchange at most c_i bits of information with $\mathtt{A}_{\text{big}}$ during an execution. Whenever $\mathtt{A}_i$ terminates its output is given to $\mathtt{A}_{\text{big}}$. The output of the computation returned by $\mathcal{S}^{(w,\,n)\text{-OC}}$ is the output of $\mathtt{A}_{\text{big}}$. Resource $\mathcal{S}^{(w,\,n)\text{-OC}}$ is parametrized by $\mathbb{P}^{\text{OC}} = \mathbb{N}^{\times 1+3n}$ where parameter $(q, \mathbf{z}, \mathbf{s}, \mathbf{c}) \in \mathbb{P}^{\text{OC}}$ with $\mathbf{z} = (z_1, \ldots, z_n)$, $\mathbf{s} = (s_1, \ldots, s_n)$ and $\mathbf{c} = (c_1, \ldots, c_n)$ denote that device i has memory s_i can leak c_i bits to $\mathtt{A}_{\text{big}}$ and contains z_i PRF keys and that a total of q calls to h (across all algorithms and all executions) can be made.

A pair $(\mathtt{naïve}, f^{()})$ is called an $(\mathsf{a}, \mathsf{b}, \epsilon)$-*one-time computable function* if it satisfies Definition 5 for $\mathcal{S}^{(w,\,n)\text{-OC}}$. A concrete scheme with bounds on its parameters is given in [42].[21]

As a new application, we very briefly describe a secure payment system with a single-message payment protocol involving a service provider and clients. The service provider generates a master PRF key k and distributes memory-constrained tamper-resistant devices (e.g. smart-cards). For example each PRF key corresponds to 1 cent. So a card worth 5 USD contains 500 PRF keys. Each card has a unique ID *ID* and the PRF keys stored in the card are generated deterministically by applying a PRF with key k to *ID*.

Clients can purchase the cards for use at a later date. In order to then pay for some amount a to obtain a service identified by a unique string S, the client uses the cards to run the NIPOE protocol in Construction 3 with target difficulty $\log_2(a)$. In more detail, they repeatedly evaluate PRFs in their cards on inputs

[21] We conjecture that their parameters carry over to the above definition using the original proof technique combined with an argument similar to that used at the end of the proof of Theorem 4.

containing S until they find an input mapping to an output prefixed by $0^{\log_2(a)}$. Upon success, they send S, the full input $(S|x)$ used in the PRF, the ID ID of the card and the index i of the PRF on that card to the service provider. The service provider verifies that evaluating a PRF with key $PRF_k(ID)$ on input $(x|S)$ is prefixed by $0^{\log_2(a)}$, and if so, accepts the transaction for service S.

One interesting but probably undesired property of the payment scheme is that the number of "burnt keys," corresponding to real-world currency, is probabilistic. The variance can be decreased by distributing the amount over multiple "smaller" proofs.

Uncomputable Functions. A closely related type of MoHF are the uncomputable functions also introduced in [42]. These have the property that they simply can not be computed on a device with less than some threshold of memory. These can also be captured by the above resource $\mathcal{S}^{(w,n)\text{-OC}}$ and parameter space $\mathbb{P}^{\text{OC}}$. However they differ in that the a and b for which security holds either return 0 or ∞, depending only on the memory-constraint parameter.

7 Interactive proofs of effort

One important practical application of MoHFs are proofs of effort (PoE), where the effort may correspond to computation, memory, or other types of resources that the hardness of which can be used in higher-level protocols to require one party, the prover, to spend a certain amount of resources before the other party, the verifier, has checked this spending and allows the protocol to continue.

7.1 Definition

Our composable definition of PoE is based on the idea of constructing an "ideal" proof-of-effort functionality from the bounded assumed resources the parties have access to in the real setting. Our Definition 5 for MoHFs can already be seen in a similar sense: from the *assumed* (bounded) resources available to the parties, evaluating the MoHF constructs a shared random function that can be evaluated for some bounded number of times. In the following, we describe the assumed and constructed resources that characterize a PoE.

The goal of PoE protocols. The high-level guarantees provided by a PoE to higher-level protocols can be described as follows. Prover P and verifier V interact in some number $n \in \mathbb{N}$ of sessions, and in each of the sessions verifier V expects to be "convinced" by prover P's spending of effort. Prover P can decide how to distribute the available resources toward convincing verifier V over the individual sessions; if prover P does not have sufficient resources to succeed in all sessions, then P can distribute its effort over the sessions. Verifier V's protocol provides as output a bit that is 1 in all sessions where the prover attributed sufficient resources, and 0 otherwise. We formalize these guarantees in the resource POE that we describe in more detail below.

Proof-of-effort resource $\mathsf{POE}^{a}_{\phi,n}$

The resource is parametrized by the numbers $n, a \in \mathbb{N}$ and a mapping $\phi : \mathbb{N} \to \mathbb{R}_{\geq 0}$. It contains as state bits $e_i, \hat{e}_i \in \{0, 1\}$ and counters $c_i \in \mathbb{N}$ for $i \in \mathbb{N}$ which are initially set to $e_i, \hat{e}_i \leftarrow 0$ and $c_i \leftarrow 0$.

Verifier V: On input a session number $i \in \{1, \ldots, n\}$, output the state e_i of that session.

Prover P:
- On input a session number $i \in \{1, \ldots, n\}$, set $c_i \leftarrow c_i + 1$. If $e_i \vee \hat{e}_i = 1$ or $\sum_{i=1}^{n} c_i > a$ then return 0. Otherwise, draw e_i (if P is honest, else $\hat{e}_i$) at random such that it is 1 with probability $\phi(c_i)$ and 0 otherwise. Output e_i (resp. $\hat{e}_i$) at interface P.
- If P is dishonest, then accept a special input copy_i that sets $e_i \leftarrow \hat{e}_i$.

The resource POE that formalizes the guarantee achieved by the PoE in a given real-world setting is parametrized by values $\underline{a}, \overline{a}, n \in \mathbb{N}$ and $\phi : \mathbb{N} \to \mathbb{R}_{\geq 0}$, and is written as $\mathsf{POE}^{\overline{a},\underline{a}}_{\phi,n} = (\mathsf{POE}^{\underline{a}}_{\phi,n}, \mathsf{POE}^{\overline{a}}_{\phi,n})$. For an honest prover P, the parameter $\underline{a} \in \mathbb{N}$ describes the overall number of "attempts" that P can take. For a dishonest prover P, the same is described by the parameter $\overline{a} \in \mathbb{N}$.[22] The success probability of a prover in each session depends on the computational resources spent in that session and can be computed as $\phi(a)$, where $a \in \mathbb{N}$ is the number of proof attempts in that session.

The "real-world" setting for PoE protocols. The PoE protocols we consider in this work are based on the evaluation of an MoHF, which, following Definition 5, can be abstracted as giving the prover and the verifier access to a shared uniform random function $\mathcal{T}^{\text{RRO}}$ that they can evaluate for a certain number of times. We need to consider both the case where the prover is honest (to formalize that the PoE can be achieved with a certain amount of resources) and the case where the prover is dishonest (to formalize that not much more can be achieved by a dishonest prover). In addition to $\mathcal{T}^{\text{RRO}}$, for n protocol sessions, the prover and verifier can also access n pairs of channels for bilateral communication, which we denote by $[\longrightarrow, \longleftarrow]^n$ in the following. (This insecure communication resource is implicit in some composable frameworks such as Canetti's UC [26].)

The resource specifies a bound $\underline{b} \in \mathbb{N}$ for the number of queries that the verifier can make to $\mathcal{T}^{\text{RRO}}$, and bounds $\underline{a}, \overline{a} \in \mathbb{N}$ for the cases where the prover is honest and dishonest, respectively. Considering the case $\underline{a} \leq \overline{a}$ makes sense because only loose bounds on the prover's available resources may be known.

The security definition. Having described the real-world and ideal-world settings, we are now ready to state the security definition. This definition will consider the above-described cases where the prover is honest (this requires that the proof

[22] For the numbers $\underline{a}, \overline{a} \in \mathbb{N}$ it may hold that $\overline{a} > \underline{a}$ because one may only know rough bounds on the available resources (at least $\underline{a}$, at most $\overline{a}$).

can be performed efficiently) and where the prover is dishonest (this requires that each proof need at least a certain effort), while we restrict our treatment to the case of honest verifiers. The security definition below follows the construction notion introduced in [62] for this specific case. The protocol and definition can additionally be extended by a hardness parameter $\mathbf{n}$ analogously to Definition 5.

Definition 10. *A protocol $\pi = (\pi_1, \pi_2)$ is a $(\phi, n, \underline{b}, \varepsilon)$-proof of effort with respect to simulator σ if for all $\underline{a}, \overline{a} \in \mathbb{N}$,*

$$\pi_1{}^P \pi_2{}^V \left[\mathcal{T}^{\mathrm{RRO}}_{\underline{a},\underline{b}}, [\longrightarrow, \longleftarrow]^n\right] \quad \approx_\varepsilon \quad \mathsf{POE}^{\underline{a}}_{\phi,n}$$

and

$$\pi_2{}^V \left[\mathcal{T}^{\mathrm{RRO}}_{\overline{a},\underline{b}}, [\longrightarrow, \longleftarrow]^n\right] \quad \approx_\varepsilon \quad \sigma^P \mathsf{POE}^{\overline{a}+n}_{\phi,n} \ .$$

The reason for the term $\overline{a}+n$ is that the dishonest prover can in each session decide to send a guess without verifying its correctness locally.

While the definition is phrased using the language of constructive cryptography [62,61], it can intuitively also be viewed as a statement in Canetti's UC framework [26].[23] For this, one would however have to additionally require the correctness formalized in the first equation of Definition 10, because UC-security would only correspond to the second equation.

7.2 Protocols

The PoE protocols we discuss in this section are interactive and start by the verifier sending a challenge to the prover, who responds with a solution. The verifier then checks this solution; an output bit signifies acceptance or rejection. There are several ways to build a scheme for PoE from an MoHF; we describe two particular schemes in this section.

Function inversion. A simple PoE can be built on the idea of having the prover invert the MoHF on a given output value. This output value is obtained by evaluating the function on a publicly known and efficiently sampleable distribution over the input space, such as the uniform distribution over a certain subset.

Construction 1. *The protocol is parametrized by a set $D \subseteq \{0,1\}^*$. For each session $1 \leq i \leq n$, it proceeds as follows:*

1. *The verifier samples $x_i \leftarrow\!\!\$\ D$, queries $y_i \leftarrow \mathcal{T}^{\mathrm{RRO}}(i, x_i)$, and sends y_i to the prover.*
2. *When activated in session i, the prover checks the next[24] possible input value $x' \in D$ for whether $\mathcal{T}^{\mathrm{RRO}}(i, x') = y_i$. If equality holds, send x' to the verifier and output 1 locally. Otherwise, output 0 locally.*

[23] One main difference is that UC is tailored toward asymptotic statements. As UC *a priori* allows the environment to create arbitrarily many instances of all protocols and functionalities, making the precise concrete statements we aim for becomes difficult.

[24] We assume that the elements in D are ordered, e.g. lexicographically.

3. *Receiving the value $x' \in D$ in session i, the verifier accepts iff $\mathcal{T}^{\text{RRO}}(i, x') = y_i$. When activated in session i, output 1 if accepted, and 0 otherwise.*

Steps 1 and 3 comprise the verifier's protocol χ, whereas step 2 describes the prover's protocol ξ. For this protocol, we show the following theorem.

Theorem 5. *Define $\zeta_j \coloneqq (|D| - j + 1)^{-1}$. If $\underline{b} > 2n$, then the described protocol (ξ, χ) is a $(\phi, n, \underline{b}, 0)$-proof of effort, with $\phi : j \mapsto \zeta_j + \frac{1-\zeta_j}{|R|}$. The simulator is described in the proof.*

Proof. We first consider the condition

$$\xi^P \chi^V \left[\mathcal{T}^{\text{RRO}}_{\underline{a},\underline{b}}, [\longrightarrow, \longleftarrow]^n\right] \quad \approx \quad \mathsf{POE}^{\underline{a}}_{\phi,n} .$$

In the following, let $d_1, d_2, \ldots \in D$ be the order in which protocol ξ iteratively tries elements of D to find a solution.

In the real-world model, in each session $i \in \{1, \ldots, n\}$ the protocol χ chooses a value $x_i \leftarrow\!\!\$\ D$, queries $y_i \leftarrow \mathcal{T}^{\text{RRO}}(i, x)$, and sends y_i to P. Upon each activation in a session i, the protocol ξ follows the strategy described in Construction 1. For the j^{th} activation in a session that has not been successful before (i.e., $\mathcal{T}^{\text{RRO}}(d_k) \neq \mathcal{T}^{\text{RRO}}(x_i)$ for $k \in \{1, \ldots, j-1\}$), and if the overall number of activations is not exceeding $\underline{a}$, the probability of $d_j = x_i$ is $\zeta_j = (|D| - j + 1)^{-1}$ as the challenge is uniform among the remaining $|D| - j + 1$ values. Additionally, if $d_j \neq x_j$, then the probability of $\mathcal{T}^{\text{RRO}}(d_j) = \mathcal{T}^{\text{RRO}}(x_i)$ is $|R|^{-1}$ since the outputs of $\mathcal{T}^{\text{RRO}}$ are uniformly distributed and independent for differing inputs. Therefore, the probability of returning 1 in the j^{th} query in a session that was not solved before is exactly $\phi(j)$, which is exactly the behavior described by $\mathsf{POE}^{\underline{a}}_{\phi,n}$. Upon activation in the i^{th} session at V, if the prover has not solved session i before, then it has not sent a message to the verifier, who outputs 0. If the prover has sent a solution $x' \in D$, then the verifier checks this via $\mathcal{T}^{\text{RRO}}(x')$ and outputs 1, as ξ sends only correct solutions. This is again the same behavior as in the case of $\mathsf{POE}^{\underline{a}}_{\phi,n}$. Overall, this means that the verifier makes at most $2n \leq \underline{b}$ queries to $\mathcal{T}^{\text{RRO}}$.

The second condition means that

$$\chi^V \left[\mathcal{T}^{\text{RRO}}_{\overline{a},\underline{b}}, [\longrightarrow, \longleftarrow]^n\right] \quad \approx \quad \sigma^P \mathsf{POE}^{\overline{a}+n}_{\phi,n}$$

with the simulator σ described as follows. Initially, it sets an internal query counter to $c \leftarrow 0$. The simulator emulates to the dishonest prover P the same interface as $[\mathcal{T}^{\text{RRO}}, [\longrightarrow, \longleftarrow]^n]$. Initially, for each session $i \in \{1, \ldots, n\}$, σ draws a uniformly random challenge $\tilde{y}_i \leftarrow\!\!\$\ R$ and makes it available on the respective (instance of the) channel $\longleftarrow$. Whenever the dishonest prover makes a query $\bar{x} \in \{0,1\}^*$ to $\mathcal{T}^{\text{RRO}}$, if $c \geq \overline{a}$ then output $\perp$. Otherwise, set $c \leftarrow c + 1$ and:

- If $\bar{x}$ has been queried before, then respond consistently.
- If $\bar{x}$ is not of the form (i, x) for $i \in \{1, \ldots, n\}$ and $x \in D$, then output a uniformly random element from R.

– If $\bar{x} = (i, x)$, and session i has not yet been solved, activate $\mathsf{POE}_{\phi,n}^{\overline{a}+n}$ in session i, and obtain a result bit b. Then, assuming this is the j^{th} (distinct) input for this session i,
  - If $b = 1$, set $g_i \leftarrow 1$ with probability[25] $\zeta_j/(\zeta_j + |R|^{-1})$ and $g_j \leftarrow 0$ otherwise. Return $\tilde{y}_i$ and mark the i^{th} session as solved.
  - Otherwise, $b = 0$. Output a uniformly random value from $R \setminus \{\tilde{y}_i\}$.
– The final case is that $\bar{x} = (i, x)$, but session i has been solved already. If $g_i = 1$, then choose a value uniformly at random from R. If $g_i = 0$, then with probability ζ_j switch $g_i \leftarrow 1$ and output $\tilde{y}_i$, and leave g_j unchanged and output a uniformly random value from R otherwise.

Once the dishonest prover sends a message $\bar{x}_i$ via the channel $\longrightarrow$ in session i:

– If a message has been sent in session i before, or $\bar{x}_i \notin D$, then ignore the message.
– If $(i, \bar{x}_i)$ has not yet been queried to $\mathcal{T}^{\text{RRO}}$, then perform the same steps as if $(i, \bar{x}_i)$ were queried as above (but do not increase c).
– If the response to query $(i, \bar{x}_i)$ was $\neq \tilde{y}_i$, then ignore the message.
– If the response to query $(i, \bar{x}_i)$ was $\tilde{y}_i$, then invoke copy_i at $\mathsf{POE}_{\phi,n}^{\overline{a}+n}$.

What remains to be shown is that with the described simulator σ, the two terms in equation Equation 7.2 are equivalent. First, the challenge values that the dishonest provers obtains on the channels $\longleftarrow$ are uniformly distributed over R in both cases. This is so by definition in the simulation, but also straightforward in the actual protocol since the values are obtained by evaluating $\mathcal{T}^{\text{RRO}}$ on distinct inputs. The responses to queries to $\mathcal{T}^{\text{RRO}}$ also have the correct distributions.

– Repeated queries are answered consistently in both cases, and queries that are not of the form (i, x) with $i \in \{1, \ldots, n\}$ and $x \in D$ are answered by a uniformly random value.
– For queries of the form $\bar{x} = (i, x_i)$, more care is necessary.
  - If the session i is not solved, and for the j^{th} (distinct) query $\bar{x}$ in session i, the probability of solving is exactly $\phi(j)$ by the same argument as in the proof of equation Equation 7.2. Consequently, in the simulation, the probability of returning $\tilde{y}_j$ is $\phi(j)$, and the probability of returning any $y \neq \tilde{y}_i$ with $y \in R$ is $(1 - \phi(j))/(|R| - 1)$.
    In the protocol, the previous attempts in the i^{th} session were obviously different from the value x_i chosen by the verifier (as otherwise the session would have been solved). The probability of returning $\tilde{y}_i$ is therefore the probability $(|D| - j + 1)^{-1}$ of guessing x_i plus the probability of not guessing x_i but still using a x' so that the output of $\mathcal{T}^{\text{RRO}}$ equals $\tilde{y}_i$, which happens with probability $(1 - (|D| - j + 1)^{-1})|R|^{-1}$. Overall, this is exactly $\phi(j)$. On the other hand, for each $y \neq \tilde{y}_i$, the probability is of *not* guessing x_i and having the output equal $y \in R$, and (by substituting

[25] This is only needed to sample correctly in sessions which have already been solved

$z := (|D| - j + 1)^{-1})$,

$$\begin{aligned}\frac{1-\phi(j)}{|R|-1} &= \left(1 - \left(z + \frac{1-z}{|R|}\right)\right) \cdot \frac{1}{|R|-1} \\ &= \left((1-z) - \frac{1-z}{|R|}\right) \cdot \frac{1}{|R|-1} \\ &= (1-z) \cdot \frac{|R|-1}{|R|} \cdot \frac{1}{|R|-1} = (1-z)|R|^{-1} \,,\end{aligned}$$

as claimed.

- In case session i has been solved previously, the output distributions can be seen to be the same as follows. The simulator σ described above manages a variable g_i per session i that tracks whether the simulator considers any input to correspond to a successful guess of x_i; this variable is set with the same probabilities that occur in the real experiment. For each subsequent input in this session, the simulator then samples uniformly if x_i is considered to be guessed, and appropriately biased toward $\tilde{y}_i$ if x_i is considered to not be guessed (and the session to instead be solved by a collision in $\mathcal{T}^{\text{RRO}}$).

Sending a value over the channels $\longrightarrow$ also has the same effects. A value that is sent in session i and is not valid, meaning that they are outside of D or known to map to a value $y \neq \tilde{y}_i$, has the effect that the verifier will never accept in session i. A value that is known to map to $\tilde{y}_i$ will make the verifier accept. A value that has not been queried to $\mathcal{T}^{\text{RRO}}$ in session i before will invoke the same sampling process as a fresh query to $\mathcal{T}^{\text{RRO}}$ in σ; in the real-world model the success probability is also the same because the probability with which $\mathcal{T}^{\text{RRO}}$ will output $\tilde{y}_i$ is the same independently of whether it is queried from P or V. This completes the proof. □

Hash trail. The idea underlying PoEs based on a hash trail is that it is difficult to compute a value such that the output of a given hash function on input this value satisfies a certain condition; usually one asks for a preimage x of a function f_i such that the output string $f_i(x) : \{0,1\}^m \to \{0,1\}^k$ starts with some number d of 0's, where $d \in \{1, \ldots, k\}$ can be chosen to adapt the (expected) effort necessary to provide a solution. For simplicity and to save on the number of parameters, we assume for the rest of the chapter that d, the hardness parameter of the moderately hard function, is also the bit-length of the output.

Construction 2. *The protocol is parametrized by sets $D, N \subseteq \{0,1\}^*$ and hardness parameter $d \in \mathbb{N}$. For each session $1 \leq i \leq n$, it proceeds as follows:*

1. *The verifier samples uniform $n_i \leftarrow\!\!\$\; N$ and sends n_i to the prover.*
2. *When activated, the prover chooses one value $x' \in D$ uniformly at random (but without collisions), computes $y \leftarrow \mathcal{T}^{\text{RRO}}(i, n_i, x_i)$, and checks whether $y[1, \ldots, d] = 0^d$. If equality holds, send x' to the verifier and output 1 locally. Otherwise, output 0 locally.*

3. *Receiving the value $x' \in D$ from the prover, the verifier accepts iff $y' \leftarrow \mathcal{T}^{\text{RRO}}(i, n_i, x')$ satisfies $y'[1, \ldots, d] = 0^d$. When activated, output 1 if the protocol has accepted and 0 otherwise.*

To capture the described scheme as a pair of algorithms (ξ, χ) as needed for our security definition, we view steps 1 and 3 as the algorithm χ, whereas step 2 describes the algorithm ξ. For this protocol, we show the following theorem.

Theorem 6. *Let $d \in \mathbb{N}$ be the hardness parameter and $\underline{b} > n$. Then the described protocol (ξ, χ) is a $(2^{-d}, \underline{b}, n, 0)$-proof of effort. The simulator σ is described in the proof.*

Proof. We first consider the condition

$$\xi^P \chi^V \left[\mathcal{T}^{\text{RRO}}_{\underline{a},\underline{b}}, [\longrightarrow, \longleftarrow]^n \right] \quad \approx \quad \mathsf{POE}^{\underline{a}}_{\phi,n} .$$

In the real-world model, in each session $i \in \{1, \ldots, n\}$ the protocol χ chooses a value $n_i \leftarrow\!\!\$\, N$ and sends n_i to P. Upon each activation in a session i, the protocol ξ follows the strategy described in Construction 2. For the j^{th} activation in a session that has not been successful before, and if the overall number of activations is not exceeding $\underline{a}$, the probability of $\mathcal{T}^{\text{RRO}}(i, n_i, x') = 0^d | y'$ for some $y' \in \{0,1\}^*$ is 2^{-d} since the outputs of $\mathcal{T}^{\text{RRO}}$ are uniformly distributed and independent for differing inputs. Therefore, the probability of returning 1 in the j^{th} query in a session that was not solved before is exactly $\phi(j)$, which is exactly the behavior described by $\mathsf{POE}^{\underline{a}}_{\phi,n}$. Upon activation in the i^{th} session at V, if the prover has not solved session i before, then it has not sent a message to the verifier, who outputs 0. If the prover has sent a solution $x' \in D$, then the verifier checks this via $\mathcal{T}^{\text{RRO}}(i, n_i, x')$ and outputs 1, as ξ sends only correct solutions. This is again the same behavior as in the case of $\mathsf{POE}^{\underline{a}}_{\phi,n}$. Overall, this means that the verifier makes at most $n \leq \underline{b}$ queries to $\mathcal{T}^{\text{RRO}}$.

The second condition means that

$$\chi^V \left[\mathcal{T}^{\text{RRO}}_{\overline{a},\underline{b}}, [\longrightarrow, \longleftarrow]^n \right] \quad \approx \quad \sigma^P \mathsf{POE}^{\overline{a}+n}_{\phi,n}$$

with the simulator σ described as follows. Initially, it sets an internal query counter to $c \leftarrow 0$. The simulator emulates to the dishonest prover P the same interface as $[\mathcal{T}^{\text{RRO}}, [\longrightarrow, \longleftarrow]^n]$. Initially, for each session $i \in \{1, \ldots, n\}$, σ draws a uniformly random value $n_i \leftarrow\!\!\$\, N$ and makes it available on the respective (instance of the) channel $\longleftarrow$. Whenever the dishonest prover makes a query $\bar{x} \in \{0,1\}^*$ to $\mathcal{T}^{\text{RRO}}$, if $c \geq \overline{a}$ then output $\perp$. Otherwise, set $c \leftarrow c + 1$ and:

- If $\bar{x}$ has been queried before, then respond consistently.
- If $\bar{x}$ is not of the form (i, n_i, x) for $i \in \{1, \ldots, n\}$, and $x \in D$, then output a uniformly random element from R.
- If $\bar{x} = (i, n_i, x)$, and session i has not yet been solved, activate $\mathsf{POE}^{\overline{a}+n}_{\phi,n}$ in session i, and obtain a result bit b. Then:
  - If $b = 1$, sample $\tilde{y}_i \leftarrow\!\!\$\, \left\{ y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^* \right\}$, return $\tilde{y}_i$ and mark the i^{th} session as solved.

- Otherwise, $b = 0$. Output a uniformly random value from

$$R \setminus \left\{ y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^* \right\}.$$

– The final case is that $\bar{x} = (i, n_i, x)$, but session i has been solved already. In that case, output a uniformly random value from R.

Once the dishonest prover sends a message $\bar{x}_i$ via the channel $\longrightarrow$ in session i:

– If a message has been sent in session i before, or $\bar{x}_i \notin D$, then ignore the message.
– If $(i, n_i, \bar{x}_i)$ has not yet been queried to $\mathcal{T}^{\text{RRO}}$, then perform the same steps as if $(i, n_i, \bar{x}_i)$ were queried as above (but do not increase c).
– If the response to query $(i, n_i, \bar{x}_i)$ was not valid (i.e., did not start with 0^d, then ignore the message.
– If the response to query $(i, n_i, \bar{x}_i)$ was valid (i.e., did start with 0^d, then invoke copy_i at $\mathsf{POE}_{\phi,n}^{\overline{a}+n}$.

What remains to be shown is that with the described simulator σ, the two terms in equation Equation 7.2 are equivalent. First, the challenge values that the dishonest provers obtains on the channels $\longleftarrow$ are uniformly distributed over R in both cases. This is so by definition in the simulation as well as in the actual protocol. The responses to queries to $\mathcal{T}^{\text{RRO}}$ also have the correct distributions.

– Repeated queries are answered consistently in both cases, and queries that are not of the form (i, n_i, x) with $i \in \{1, \ldots, n\}$, n_i as output in the i^{th} session before, and $x \in D$ are answered by a uniformly random value.
– For queries of the form $\bar{x} = (i, n_i, x_i)$ with $x_i \in D$, more care is necessary.
  - If the session i is not solved, and for the j^{th} (distinct) query $\bar{x}$ in session i, the probability of solving is exactly $\phi(j) = 2^{-d}$ by the same argument as in the proof of equation Equation 7.2. Consequently, in the simulation, the probability of returning a (uniformly random) value from the set $S = \left\{ y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^* \right\}$ is 2^{-d}, and the probability of returning a (uniformly random) value from $R \setminus S$ is $(1 - \phi(j)) = 1 - 2^{-d}$. As $|R|/|S| = 2^d$, the resulting output is uniformly random in R.
    In the protocol, since $\mathcal{T}^{\text{RRO}}$ has never been queried on the input before, the output is uniformly random from R by definition of $\mathcal{T}^{\text{RRO}}$.
  - In case session i has been solved previously, the output is uniformly random from R in both cases.

Sending a value over the channels $\longrightarrow$ also has the same effects. A value that is sent in session i and is not valid, meaning that it is outside of D or known to not map to a valid solution, has the effect that the verifier will never accept in session i. A value that is known to map to a valid solution will make the verifier accept. A value that has not been queried to $\mathcal{T}^{\text{RRO}}$ in session i before will invoke the same sampling process as a fresh query to $\mathcal{T}^{\text{RRO}}$ in σ; in the real-world model the success probability is also the same because the probability with which $\mathcal{T}^{\text{RRO}}$ will output a valid value is the same independently of whether it is queried from P or V. This completes the proof. □

8 Non-interactive proofs of effort

The PoE protocols in Section 7.2 require the prover and the verifier to interact, because the verifier has to generate a fresh challenge for the prover in each session to prevent the prover from re-using (parts of) proofs in different sessions. This interaction is inappropriate in several settings, because it either imposes an additional round-trip on protocols (such as in key establishment) or because a setting may be inherently non-interactive, such as sending e-mail. In this section, we describe a non-interactive variant of PoE that can be used in such scenarios. Each proof is cryptographically bound to a certain value, and the higher-level protocol has to make sure that this value is bound to the application so that proofs cannot be re-used.

Although non-interactive PoE (niPoE) have appeared previously in certain applications, and have been suggested for fighting spam mail [1,37,38,39], to the best of our knowledge they have not been formalized as a tool of their own right.

8.1 Definition

Our formalization of non-interactive PoE (niPoE) follows along the same lines as the one for the interactive proofs. The main difference is that while for interactive proofs, it made sense to some notion of session to which the PoE is associated and in which the verifier sends the challenge, this is not the case for niPoE. Instead, we consider each niPoE as being bound to some particular *statement* $s \in \{0,1\}^*$. This statement s is useful for binding the PoE to a particular context: in the combatting-spam scenario this could be a hash of the message to be sent, in the DoS-protection for key exchange this could be the client's key share.

For consistency with Section 7, the treatment in this section is simplified to deal with either only honest or only dishonest provers. The case where both honest and dishonest provers occur simultaneously is deferred to Appendix C.

The goal of niPoE protocols. The constructed resource is similar to the resource POE described in Section 7.1, with the main difference that each proof is not bound to a session $i \in \mathbb{N}$, but rather to a statement $s \in \mathcal{S} \subseteq \{0,1\}^*$. Consequently, the resource NIPOE takes as input at the P-interface statements $s \in \mathcal{S}$, and returns 1 if the proof succeeded and 0 otherwise. Upon an activation at the verifier's interface V, if for any statement $s \in \mathcal{S}$ a proof has been successful, the resource outputs this s, and it outputs $\perp$ otherwise. An output $s \neq \perp$ has the meaning that the party at the P-interface has spent enough effort for the particular statement s. Similarly to POE, the resource NIPOE is parametrized by a bound $a \in \mathbb{N}$ on the number of proof attempts and a performance function $\phi : \mathbb{N} \to \mathbb{R}_{\geq 0}$, but additionally the number of verification attempts $\underline{b} \in \mathbb{N}$ at the verifier is a parameter. The resource is denoted as $\mathsf{NIPOE}^{a}_{\phi,\underline{b}}$. The behavior of this resource is described in more formally below. There are two inputs for a dishonest prover P that need further explanation:

– (copy, s): This corresponds to sending a proof to V. Prover V is convinced if the proof was successful (i.e., $e_s = 1$), and has to spend one additional evaluation of $\mathcal{T}^{\text{RRO}}$, so the corresponding counter is increased ($d \leftarrow d + 1$).
– (spend): E forces V to spend one additional evaluation of $\mathcal{T}^{\text{RRO}}$, for instance by sending an invalid proof. This decreases the number of verifications that V can still do ($d \leftarrow d + 1$).

Non-interactive proof-of-effort resource $\mathsf{NIPOE}^{a}_{\phi,\underline{b}}$

The resource is parametrized by numbers $a, \underline{b} \in \mathbb{N}$ and a mapping $\phi : \mathbb{N} \to \mathbb{R}_{\geq 0}$. It contains as state bits $e_s \in \{0,1\}$ and counters $d, c_s \in \mathbb{N}$ for each $s \in \{0,1\}^*$ (all initially 0), and a list $\mathsf{S} \in (\{0,1\}^*)^*$ of strings that is initially empty.

Verifier V: On input a unary value, if S is empty then return $\perp$. Otherwise remove the first element of S and return it.

Honest prover P: On input a string $s \in \{0,1\}^*$, set $c_s \leftarrow c_s + 1$. If $e_s = 1$ or $\sum_{s \in \{0,1\}^*} c_s > a$, then return 0. Otherwise, draw e_s at random such that it is 1 with probability $\phi(c_s)$ and 0 otherwise. If $e_s = 1$ and $d < \underline{b}$, then $d \leftarrow d + 1$ and then append s to S. Output e_s at interface P.

Dishonest prover P:
– On input a string $s \in \{0,1\}^*$, set $c_s \leftarrow c_s + 1$. If $e_s = 1$ or $\sum_{s \in \{0,1\}^*} c_s > a$, then return 0. Otherwise, draw e_s at random such that it is 1 with probability $\phi(c_s)$ and 0 otherwise. Output e_s at interface P.
– Upon an input (copy, s), if $d < \underline{b}$ and $e_s = 1$, then $d \leftarrow d + 1$ append s to S.
– Upon an input (spend), set $d \leftarrow d + 1$.

The "real-world" setting for niPoE protocols. The main difference between PoE and niPoE is that a PoE requires bidirectional communication, which in Section 7.1 we described by the channels $\longrightarrow$ and $\longleftarrow$ available in each session. A niPoE only requires communication from the prover to the verifier, which we denote by the channel $\longrightarrow$. Additionally, and as in the PoE case, the proof also requires computational resources, which are again formalized by the shared resource $\mathcal{T}^{\text{RRO}}_{\underline{a},\underline{b}}$.

The security definition. The definition of niPoE security is analogous to the one for PoE.

Definition 11. *A protocol $\pi = (\pi_1, \pi_2)$ is a* non-interactive $(\phi, \underline{b}, \varepsilon)$-proof-of-effort with respect to simulator σ *if for all $\underline{a}, \overline{a} \in \mathbb{N}$,*

$$\pi_1{}^P \pi_2{}^V \left[\mathcal{T}^{\text{RRO}}_{\underline{a},\underline{b}}, \longrightarrow\right] \quad \approx_\varepsilon \quad \mathsf{NIPOE}^{\underline{a}+\underline{b}}_{\phi,\underline{b}}$$

and

$$\pi_2{}^V \left[\mathcal{T}^{\text{RRO}}_{\overline{a},\underline{b}}, \longrightarrow\right] \quad \approx_\varepsilon \quad \sigma^P \mathsf{NIPOE}^{\overline{a}+\underline{b}}_{\phi,\underline{b}} .$$

8.2 Protocol

Our protocol for niPoE is similar to the one in Construction 2. Instead of binding the solution to a session identifier chosen by the server, however, the identifier

is chosen by the client. This makes sense for instance in the setting of sending electronic mail where the PoE can be bound to a hash of the message, or in Denial-of-Service protection in the TLS setting, where the client can bind the proof to its ephemeral key share.[26]

Construction 3. *The protocol is parametrized by sets $D, \mathcal{S} \subseteq \{0,1\}^*$ and a hardness parameter $d \in \mathbb{N}$. It proceeds as follows:*

1. *On input a statement $s \in \mathcal{S}$, the prover chooses $x \in D$ uniformly at random (but without collisions with previous attempts for the same s), computes $y \leftarrow \mathcal{T}^{\text{RRO}}(s, x)$, and checks whether $y[1, \ldots, d] = 0^d$. If equality holds, send (s, x, y) to the verifier and output 1 locally, otherwise output 0.*
2. *Upon receiving $(s', x', y) \in \mathcal{S} \times D \times R$, the verifier accepts s iff $y' \leftarrow \mathcal{T}^{\text{RRO}}(s', x')$ satisfies $y = y'$ and $y'[1, \ldots, d] = 0^d$. If the protocol is activated by the receiver and there is an accepted value $s' \in \mathcal{S}$, then output s'.*

To capture the described scheme as a pair of converters (ξ, χ) as needed for our security definition, we view step 2 as the converter χ, whereas step 1 describes the converter ξ. For this protocol, we show the following theorem.

Theorem 7. *Let $d \in \mathbb{N}$ the hardness parameter. Then the described protocol (ξ, χ) is a non-interactive $(2^{-d}, \underline{b}, 0)$-proof-of-effort.*

Proof. We first consider the condition

$$\xi^P \chi^V \left[\mathcal{T}^{\text{RRO}}_{\underline{a},\underline{b}}, \longrightarrow\right] \quad \approx \quad \mathsf{NIPOE}^{\underline{a}}_{\phi,\underline{b}} \, .$$

In the real-world model, upon each input $s \in \mathcal{S}$ at P, the protocol ξ follows the strategy described in Construction 2. For each activation for $s \in \mathcal{S}$ for which P has not been successful before, and if the overall number of activations is not exceeding $\underline{a}$, the probability of $\mathcal{T}^{\text{RRO}}(s, x') = 0^d | y'$ with $y' \in \{0,1\}^*$ is 2^{-d} since the outputs of $\mathcal{T}^{\text{RRO}}$ are uniformly distributed and independent for differing inputs. Therefore, the probability of returning 1 in the j^{th} query in a session that was not solved before is exactly $\phi(j)$, which is exactly the behavior described by $\mathsf{NIPOE}^{\underline{a}}_{\phi,\underline{b}}$. Upon an activation at V, if the prover has not solved for any (new) statement, then it has not sent a message to the verifier, who outputs $\perp$. If the prover has sent a solution $(s', x') \in \mathcal{S} \times D$, then the verifier checks this via $\mathcal{T}^{\text{RRO}}(s', x')$ and outputs s', as ξ sends only correct solutions, for up to $\underline{b}$ statements. This is again the same behavior as in the case of $\mathsf{NIPOE}^{\underline{a}}_{\phi,\underline{b}}$.

The second condition means that

$$\chi^V \left[\mathcal{T}^{\text{RRO}}_{\overline{a},\underline{b}}, \longrightarrow\right] \quad \approx_\varepsilon \quad \sigma^P \mathsf{NIPOE}^{\overline{a}}_{\phi,\underline{b}} \, ,$$

with $\varepsilon = \frac{\underline{b} \cdot 2^d}{|R|}$ and the simulator σ described as follows. Initially, it sets internal query counters to $c_P, c_V \leftarrow 0$. The simulator emulates to the dishonest prover P the same interface as $\left[\mathcal{T}^{\text{RRO}}_{\overline{a},\underline{b}}, \longrightarrow\right]$. Whenever the dishonest prover makes a query $\bar{x} \in \{0,1\}^*$ to $\mathcal{T}^{\text{RRO}}$, if $c_P \geq \overline{a}$ then output $\perp$. Otherwise, set $c_P \leftarrow c_P + 1$ and:

[26] This works especially well with the new ordering in TLS 1.3.

– If $\bar{x}$ has been queried before, then respond consistently.
– If $\bar{x}$ is not of the form $(s, x) \in \mathcal{S} \times D$, then output a uniformly random element from R.
– If $\bar{x} = (s, x) \in \mathcal{S} \times D$, and for statement s the prover has not yet been solved, activate $\mathsf{NIPOE}^{\overline{a}}_{\phi,\underline{b}}$ with statement s, and obtain a result bit b. Then:
  • If $b = 1$, sample $\tilde{y}_i \leftarrow\!\!\$ \left\{y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^*\right\}$, return $\tilde{y}_i$ and mark s as solved.
  • Otherwise, $b = 0$. Output a uniformly random value from

$$R \setminus \left\{y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^*\right\}.$$

– The final case is that $\bar{x} = (s, x) \in \mathcal{S} \times D$, but the prover has solved for statement s already. In that case, output a uniformly random value from R.

Whenever the dishonest prover sends a message $\bar{x}$ via the channel $\longrightarrow$:

– If $\bar{x} \notin \mathcal{S} \times D \times R$, or $c_V \geq \underline{b}$, then ignore the message. Otherwise, set $c_V \leftarrow c_V + 1$.
– If $\bar{x} = (s, x, y)$ but the response to query (s, x) was not valid (i.e., did not start with 0^d) or not equal to y, then input (spend) to $\mathsf{NIPOE}^{\overline{a}}_{\phi,\underline{b}}$.
– If $\bar{x} = (s, x, y)$ and the response to query (s, x) was y, and y starts with 0^d, then invoke copy_s at $\mathsf{NIPOE}^{\overline{a}}_{\phi,n}$.
– If $\bar{x} = (s, x, y)$ but (s, x) has not yet been queried to $\mathcal{T}^{\text{RRO}}$, then behave as if (s, x) where queried to $\mathcal{T}^{\text{RRO}}$ and continue appropriately as in the two above cases.

What remains to be shown is that with the described simulator σ, the two terms in equation Equation 7.2 are equivalent. First, the responses to queries to $\mathcal{T}^{\text{RRO}}$ also have the correct distributions.

– Repeated queries are answered consistently in both cases, and queries that are not of the form $(s, x) \in \mathcal{S} \times D$ are answered by a uniformly random value.
– For queries of the form $\bar{x} = (s, x) \in \mathcal{S} \times D$, more care is necessary.
  • If the prover has not yet solved for statement s, the probability of solving is exactly $\phi(j) = 2^{-d}$ by the same argument as in the proof of equation Equation 8.2. Consequently, in the simulation, the probability of returning a (uniformly random) value from the set

$$S = \left\{y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^*\right\}$$

  is 2^{-d}, and the probability of returning a (uniformly random) value from $R \setminus S$ is $(1 - \phi(j)) = 1 - 2^{-d}$. As $|R|/|S| = 2^d$, the resulting output is uniformly random in R.
  In the protocol, since $\mathcal{T}^{\text{RRO}}$ has never been queried on the input before, the output is uniformly random from R by definition of $\mathcal{T}^{\text{RRO}}$.
  • In case session i has been solved previously, the output is uniformly random from R in both cases.

Sending a value over the channel $\longrightarrow$ has the same effects. A value that is not valid, meaning that it is outside of $\mathcal{S} \times D \times R$ or known to not map to a valid solution, does not change the state of the verifier. A value that is known to map to a valid solution will make the verifier accept the corresponding statement. A message (s, x, y) where (s, x) has not been queried to $\mathcal{T}^{\text{RRO}}$ before will be treated analogously to a query to $\mathcal{T}^{\text{RRO}}$ and subsequently sending the result in both cases. This completes the proof. □

9 Combining the results

Before we can compose the MoHFs proven secure according to Definition 5 with the application protocols described in Sections 7 and 8 using the respective composition theorem [62,61], we have to resolve one apparent incompatibility. The indifferentiability statement according to Definition 5 is not immediately applicable in the case with two honest parties, as required in the availability conditions of Definitions 10 and 11, where both the prover and verifier are honest.[27] We further explain how to resolve this issue in Appendix A; the result is that for stateless algorithms, Definition 5 immediately implies the analogous statement for resources with more honest interfaces, written $\mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}}$ and $\mathcal{T}^{\text{RRO}}_{a_1,a_2,b}$, which have two "honest" interfaces priv_1 and priv_2.

We can then immediately conclude the following corollary from composition theorem [62,61] by instantiating it with the schemes of Definitions 5 and 10. An analogous corollary holds for the niPoEs.

Corollary 1. *Let* $f^{(\cdot)}, \mathtt{naïve}, \mathbb{P}, \pi, \mathsf{a}, \mathsf{b} : \mathbb{P} \to \mathbb{N}$, *and* $\varepsilon : \mathbb{P} \times \mathbb{P} \to \mathbb{R}_{\geq 0}$ *as in Definition 5, and let* (ξ, χ) *be a* $(\phi, n, \underline{b}, \varepsilon')$*-proof of effort. Then*

$$\xi^P \chi^V \left[\pi^P \pi^V \bot^{\text{pub}} \mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}}, [\longrightarrow, \longleftarrow]^n\right] \quad \approx_\varepsilon \quad \mathsf{POE}^{\mathsf{a}(\mathbf{l}_1)}_{\phi,n} ,$$

with $P = \text{priv}_1$ *and* $V = \text{priv}_2$*, for all* $\mathbf{l}_1, \mathbf{l}_2 \in \mathbb{P}$*, and where* $\bot^{\text{pub}} \mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}}$ *means that the* pub*-interface is not accessible to the distinguisher. Additionally,*

$$\chi^V \left[\pi^V \bot^{\text{priv}_1} \mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}}, [\longrightarrow, \longleftarrow]^n\right] \quad \approx_\varepsilon \quad \tilde{\sigma}^P \mathsf{POE}^{\mathsf{b}(\mathbf{r})+n}_{\phi,n} ,$$

with $P = \text{pub}$ *and* $V = \text{priv}_2$*, for all* $\mathbf{r}, \mathbf{l}_2 \in \mathbb{P}$*, and where* $\tilde{\sigma}$ *is the composition of the two simulators guaranteed by Definitions 5 and 10.*

10 Detailed discussion of related work and applications

10.1 Related work

Definitions of moderately hard functions. Existing notions of moderately hard functions [68,44,19,11,4] essentially define a game in which an adversary has to evaluate the function in the forward direction. For many applications, such as

[27] The verifier is always considered honest in our work.

the most natural PoE constructions based on finding inputs to the MoHF such that its output has some desired property, it is important for the MoHF to also be difficult to *invert*. These applications only indirectly require the difficulty of computing it in the forward direction.[28] As we show in the applications, our definition is indeed sufficient for proving security of those constructions.

The definitions that are conceptually most similar to ours are those of Bellare *et al.* [17] and Demay *et al.* [35]. Both papers discuss hash-function iteration as a method to increase the hardness of the function, and also present a definition based on indifferentiability. Their definitions, however, are restricted to the complexity measure of counting the number of random-oracle invocations. Our work is based on the same framework, but shows how more general hardness notions can be captured.

Candidate constructions. The seminal paper of Dwork and Naor [38] discusses several candidate constructions based on supposedly moderately hard-to-compute mathematical problems. Most subsequent work is based on properties of hash functions, such as using the plain hash function [2] or iterating the function to increase the hardness of inverting it. Iteration seems to first appear in the Unix crypt function [65] and analyzed by Yao and Yin [82] and Bellare *et al.* [17], a prefixing scheme has been discussed and analyzed by Demay *et al.* [35].

Another notion of hardness is based on forcing the processor to access the (slower) main memory because the data needed to compute the functions do not fit into the (small) cache. Such functions were proposed in the context of combatting spam mail [1,37,39], and the rough idea is that during the computation of the function one has to access various position in a random-looking array that is too large to fit into cache. We discuss the reduction that will be necessary to make those functions useful in our framework in Section 6.

A more recent popular approach draws its inspiration from the work of Percival [68]. These MoHFs rely on a notion of hardness aimed at ensuring that application-specific integrated circuits (ASICs) have as little advantage (in terms of dollar per rate of computation) over general-purpose hardware. This is motivated by the observation that, both for large-scale password cracking and for high-capacity brute-forcing of the PoEs underlying crypto-currencies, the go-to computational device in practice has often been a (highly parallel) ASIC. In the case of password cracking, this results in undesirably efficient cracking devices while for crypto-currencies the effect is to convey an disproportionate amount of influence from a comparatively small subset of users that are willing to invest in such hardware; an undesirable effect in what are meant to be egalitarian distributed systems. Percival had the insight that high-speed memory for ASICs is comparatively very expensive while general purpose computers are usually equipped with large amounts of cheap DRAM. Therefore, he proposed [68] building functions which require large amounts of memory to compute. The idea has been well received and [68] has been followed by a large number of proposed

[28] Intuitively it is also the difficulty of inverting the MoHF which makes it useful for the password storage applications proposed in [11,44].

constructions [44,11,23,32,70,53,81,7]. A notable example is `Argon2` [18], an early version of which was the winner of the Password Hashing Competition.

The most prominent application of memory-hard functions thus far has been password hashing. Here, rather than storing a password in the clear, a login server will evaluate a memory-hard function on users passwords and store the result. That way, by recomputing the function at the next login the server can still authenticate users. But if an adversary breaks in and steals the password file they are faced with the task of inverting the outputs of the function in order to recover valid login credentials. Another application are PoEs such as those in Litecoin [28], Dogecoin [59] and several other crypto-currencies. In general in all proposed and actual applications, an adversary is ultimately faced with the task of inverting the memory-hard function.[29]

The first security notion of memory-hard functions was given in [68] which asks for a lower bound on the product of memory and time used by an algorithm evaluating the function on any single input. Another step towards provably secure applications was made in [11] where a concrete computational model—the parallel random oracle model (pROM)—was introduced together with a computational notion of complexity called amortized Cumulative Memory Complexity (aCMC). This notion calls for a lower bound on the *amortized* product of space and time across an arbitrary number of evaluations. aCMC was further refined in [4] to account for possible trade-offs of decreasing memory consumption at the added cost of increased logic gates resulting in the notion of amortized Energy Complexity (aEC). For simplicity, in this work we focus on aCMC as the techniques carry over directly to the more complicated case of aEC. The pROM was refined in [9] to better capture the nature of computation using an ASIC and it is that version which serves as a starting point for our definition of memory-hard functions. (For more on the choices for the complexity notion and computational model in this work we refer to the remarks after Theorem 4.)

We remark that, an important shortcoming of aCMC (and eEC) is that they only consider the (amortized) complexity of evaluating an MHF in the forward direction. Thus notions of MoHFs in those works can serve, at best, as the basis of heuristic arguments for the security of their applications in practice. Another problem standing in the way of basing provable security on aCMC is that is defined to only bound the *expected* complexity. Yet this is too weak for any reasonable security application as expectation can be dominated by rare outlier cases. In other words, large expected complexity does not rule out very low complexity on the vast number of instances but extremely high complexity on very rare instances; an undesirable property for most applications of MoHFs.

Nevertheless, recent results [8,9] raise hopes that even `scrypt` may be provable in our framework, although our techniques in Theorem 4 does not apply.

Proofs of effort. Proofs of effort have been first introduced (informally) in the work of Dwork and Naor [38] on combatting spam mail and a line of follow up

[29] E.g in the case of a PoE the goal is to find an input which maps to any element in a large set of possible outputs.

works [1,37,39], but have found applications in various scenarios. One particular widely-discussed one is in client-server applications, where Denial-of-Service attacks can be launched on the server by forcing it to perform costly cryptographic computations. The high-level idea of the moderately hard so-called "client puzzles" in this domain is that the client has to perform some computation in the very beginning of a session, rendering Denial-of-Service attacks infeasible. Applications in the area of key-agreement and secure-session protocols have been described in the literature [54,33,76]. A formal definition of client puzzles appeared in the work of Chen *et al.* [29], but has later been shown to be insufficient for the case of multiple sessions and improved by Stebila *et al.* [75], which has subsequently been strengthened by Groza and Warinschi [49].

Initiated by Bitcoin [66], several crypto-currencies have emerged in recent years. The underlying principle is that all transactions are recorded in a public ledger. Extending this ledger is designed to be computationally feasible but expensive (it is intuitively based on moderately hard functions); this results in a scheme that is secure as long as no dishonest party obtains the majority in terms of computational power [45]. Other blockchains have been designed that are not based on computation hardness but rather on memory hardness [28,24]. This area has spawned a lot of interest in the cryptographic community, recent results showing that the underlying technique can also be used in the design multi-party protocols whose security is based on the majority of computational power [12]. A related concept is *resource fairness* as proposed by Garay *et al.* [47], which assures that in a multi-party computation all parties can receive their output by investing a similar amount of computational resources.

Another definition of proof of effort appears in the work of Alberini *et al.* [3]. The definition, there called *private proof of effort* is somewhat similar to our Definition 10 in Section 7, but for very specific parameters, as in their definition the prover can solve an instance by (exactly) one call to the so-called "effort oracle." Our definition is more general and allows more fine-grained statements about (the widely used) protocols in which the prover can take multiple attempts. Stand-alone definitions for proofs of space have recently been given by Ateniese *et al.* [13] and Dziembowski *et al.* [40].

Further applications. The need for moderately hard computation has also appeared in partial key escrow [16], where the key escrow is implemented in a way such that actually obtaining a key is computationally challenging but feasible for authorities. This allows to obtain the keys of specific targets, while at the same time hindering large-scale surveillance. While the moderately hard functions we describe in this cannot directly be used in the known constructions, a more detailed study appears promising.

Another related area is that of time-lock puzzles, which has started with the work of Rivest *et al.* [73]. The idea is to design computations such that they can be performed in a pre-determined span of real-time; this has applications such as time-stamping or delayed key escrow, but also proofs of work. For time-lock puzzles it is particularly important that the computation cannot be parallelized.

Several constructions of time-lock puzzles have been proposed, based on different assumptions [79,51,55,77,58,57,22].
Various further applications have been discussed in an invited talk by Naor [67].

10.2 Applications

In this section, we further discuss the usefulness of our definitions in application scenarios that have been studied in the literature.

Combatting spam mail. A line of work initiated by Dwork and Naor [38] and continued by [1,37,39] proposes the use of moderately hard functions to fight spam mail. According to their ideas, sending an email requires solving a computational problem based on a computationally hard function, a concept that we called a *non-interactive proof of effort* in Section 8. In a nutshell, and using the notation from [39], their scheme works as follows:

- Compute $A \leftarrow H(m, R, S, d, k)$ with message m, receiver R, sender S, date d, and auxiliary value k.
- Evolve A by computing a (randomized) graph function that incurs some $\ell \in \mathbb{N}$ calls to hash functions.
- Accept if the result of applying a hash function H' to the result of the graph function has d trailing zeroes, for hardness parameter d. Send the output as well as k as a proof of effort.

The main result, [39, Theorem 1], is that the amortized complexity of a spammer of generating a proof that will be accepted by the verifier is $\Omega(2^d \ell)$, where the asymptotic statement is over the number of proofs.

Analogous results can be obtained using our framework. In particular, use of the resource NIPOE described in Section 8 makes deriving the statement very easy: we simply require the sender of a mail to provide a proof associated to the statement (m, R, S, d).[30] Analogously to [37, Lemma 1], one easily obtains that amortized over the number of proofs the (dishonest) prover has to activate NIPOE for $\Omega(2^d)$ times. Once we instantiate NIPOE with an MoHF that requires cost $\ell \in \mathbb{N}$ per evaluation, we immediately obtain the bound of $\Omega(2^d)$ simply by application of the composition theorem.

Consequently, *any* MoHF that satisfies our definition immediately gives rise to a scheme for combatting spam mail along the lines of [1,37,39].

Client puzzles and Denial-of-Service resilience. The theoretical groundwork for the use of client puzzles in Denial-of-Service resilience has been laid by [29,75,49]. The work of Chen *et al.* [29] considered the hardness only for a single, isolated session. The works of Stebila *et al.* [75] and Groza and Warinschi [49] measure the hardness of multiple parallel sessions. Their definition is roughly similar to our POE-security in Section 7, the main differences are that [49] considers the

[30] When instantiating NIPOE with an appropriate MoHF, the resulting scheme is similar to the ones in [1,37,39].

probability of an adversary in solving *all* sessions, whereas our definition is more fine-grained and considers the probabilities per-session. On the other hand, the adversary in their definition has access to an oracle providing it with properly solved puzzles (which is not the case in our definition).[31] The statements and proofs in [49] only require that the adversary algorithm accesses the MoHF as an oracle, the results can be re-phrased in different computational models and make use of MoHFs proven secure in our work, bringing up client puzzles for other notions of hardness. These puzzles can then also be used to achieve Denial-of-Service resilience via the protocols of Stebila *et al.* [75].

Public transaction ledgers based on Nakamoto consensus. The work of Garay *et al.* [45] studies properties of a public transaction ledger that is managed through the Nakamoto consensus, and related, protocols, which are based on proof of work. The properties are formalized in a multi-party computation model based on Canetti's work [25]. The model is synchronous and during each round the (honest and corrupted) parties are activated in a round-robin fashion. The hardness of the proof of work is, as in the examples discussed above, modeled by restricting the number of queries that each party (honest or corrupted) can make to a random oracle. Unlike in the above two cases, instantiating this random oracle using an MoHF proven according to Definition 5 does *not* make sense, as the model allows the adversary only to run a single, non-interactive algorithm.

Interestingly, the hardness amplification statement of Demay *et al.* [35], which is phrased in a similar, indifferentiability-like fashion, does carry over also to that statement. The "real-world" resources do not restrict the way in which they can be used by the adversary, and the simulator reacts to the adversaries queries in a fully on-line way. Previous works [17,35] consider only a single hardness notion, namely the number of queries that a party makes to a random oracle, for which the environment (or distinguisher) does not have "implicit" resources.

An interesting open question is therefore to either generalize our definition or modify the model of [45] in a way that allows for proving security also based on other hardness assumptions (such as memory-hard or memory-bound functions). One possible idea of restricting the model of [45] would be to use a different RO instance per round (which can be argued because the prefix is only known after the previous round is finished) and to restrict the attention to miners that do not use messages received in the same round for mining their own block in this round (which appears a reasonable assumption). We leave this as an interesting (and pressing) question for future work.

11 Open Questions

We discuss several interesting open questions raised by this work. The topic of moderately hard functions is an active topic of research both in terms of defi-

[31] Such an oracle is inherently not useful if one demands that the difficulty of solving n puzzles in parallel is (approximately) n times the difficulty of solving a single puzzle, which the schemes we consider achieve for realistic parameters.

nitions and constructions and so many practically interesting (and used) moderately hard function constructions and proof-of-effort protocols could benefit from a more formal treatment (e.g. Equihash [20], CryptoNight, Ethash). Many of these will likely result in novel instantiates of the MoHF framework which we believe to be of independent interest as this requires formalizing new security goals motivated by practical considerations. In terms of new moderately hard functions, the recent work of Biryukov and Perrin [21] introduces several new constructions for use in hardening more conventional cryptographic primitives against brute-force attacks. For this type of application, a composable security notion of moderate hardness such as the one in this work would lend itself well to analyzing the effect on the cryptographic primitives being hardened. Other examples of recent proof-of-effort protocols designed to for particular higher-level applications in mind are the results in [15,30,43,46]. In each case, at most standalone security of the higher-level application can be reasoned about so using the framework in this paper could help improve the understanding of the applications composition properties.

A natural question that arises from how the framework is currently formulated is whether the ideal-world resource could be relaxed. While modeling the ideal resource as a random oracle does make proving security for applications using the MoHF easier it seems to moot ever proving security for any candidate MoHF outside the random oracle model. However, it would be nice to show some form of moderate hardness based on other assumptions or, ideally, even unconditionally. Especially in the domain of client-puzzles several interesting constructions already exists based on various computational hardness assumptions [74,55,79,52].

References

1. Abadi, M., Burrows, M., Manasse, M., Wobber, T.: Moderately hard, memory-bound functions. ACM Trans. Internet Technol. 5(2), 299–327 (2005)
2. Adam Back: Hashcash - A Denial of Service Counter-Measure (2002)
3. Alberini, G., Moran, T., Rosen, A.: Public verification of private effort. In: Dodis, Y., Nielsen, J.B. (eds.) TCC. LNCS, vol. 9015, pp. 169–198 (2015)
4. Alwen, J., Blocki, J.: Efficiently computing data-independent memory-hard functions. In: CRYPTO. LNCS, vol. 9815, pp. 241–271 (2016)
5. Alwen, J., Blocki, J.: Towards Practical Attacks on Argon2i and Balloon Hashing. In: EuroS&P 2017 (2017)
6. Alwen, J., Blocki, J., Harsha, B.: Practical graphs for optimal side-channel resistant memory-hard functions. Cryptology ePrint Archive, Report 2017/443 (2017), `http://eprint.iacr.org/2017/443`
7. Alwen, J., Blocki, J., Pietrzak, K.: Depth-robust graphs and their cumulative memory complexity. In: EUROCRYPT. LNCS (2017), `https://eprint.iacr.org/2016/875`
8. Alwen, J., Chen, B., Kamath, C., Kolmogorov, V., Pietrzak, K., Tessaro, S.: On the complexity of Scrypt and proofs of space in the parallel random oracle model. In: EUROCRYPT. LNCS, vol. 9666, pp. 358–387 (2016)

9. Alwen, J., Chen, B., Pietrzak, K., Reyzin, L., Tessaro, S.: Scrypt is maximally memory-hard. In: EUROCRYPT. LNCS (2017)
10. Alwen, J., Gaži, P., Kamath, C., Klein, K., Osang, G., Pietrzak, K., Reyzin, L., Rolínek, M., Rybár, M.: On the Memory-Hardness of Data-Independent Password-Hashing Functions. Cryptology ePrint Archive, Report 2016/783 (2016)
11. Alwen, J., Serbinenko, V.: High Parallel Complexity Graphs and Memory-Hard Functions. In: STOC (2015)
12. Andrychowicz, M., Dziembowski, S.: Distributed cryptography based on the proofs of work. Cryptology ePrint Archive, Report 2014/796 (2014)
13. Ateniese, G., Bonacina, I., Faonio, A., Galesi, N.: Proofs of space: When space is of the essence. In: SCN. pp. 538–557 (2014)
14. Aura, T., Nikander, P., Leiwo, J.: Dos-resistant authentication with client puzzles. In: Security Protocols, LNCS, vol. 2133, pp. 170–177 (2001)
15. Ball, M., Rosen, A., Sabin, M., Vasudevan, P.N.: Proofs of useful work. Cryptology ePrint Archive, Report 2017/203 (2017), http://eprint.iacr.org/2017/203
16. Bellare, M., Goldwasser, S.: Verifiable partial key escrow. In: CCS. pp. 78–91 (1997)
17. Bellare, M., Ristenpart, T., Tessaro, S.: Multi-instance security and its application to password-based cryptography. In: CRYPTO, LNCS, vol. 7417, pp. 312–329 (2012)
18. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: New generation of memory-hard functions for password hashing and other applications. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016. pp. 292–302. IEEE (2016), http://dx.doi.org/10.1109/EuroSP.2016.31
19. Biryukov, A., Khovratovich, D.: Tradeoff cryptanalysis of memory-hard functions. In: ASIACRYPT. pp. 633–657 (2015)
20. Biryukov, A., Khovratovich, D.: Equihash: Asymmetric proof-of-work based on the generalized birthday problem. Ledger Journal 2 (2017)
21. Biryukov, A., Perrin, L.: Symmetrically and asymmetrically hard cryptography (full version). Cryptology ePrint Archive, Report 2017/414 (2017), http://eprint.iacr.org/2017/414
22. Bitansky, N., Goldwasser, S., Jain, A., Paneth, O., Vaikutanathan, V., Waters, B.: Time-lock puzzles from randomized encodings. Cryptology ePrint Archive, Report 2015/514 (August 2015)
23. Boneh, D., Corrigan-Gibbs, H., Schechter, S.E.: Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In: ASIACRYPT. LNCS, vol. 10031, pp. 220–248 (2016)
24. Buterin, V., Di Lorio, A., Hoskinson, C., Alisie, M.: Ethereum: A Distributed Cryptographic Leger (2013), http://www.ethereum.org/
25. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology 13(1), 143–202 (2000)
26. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)
27. Canetti, R., Halevi, S., Steiner, M.: Hardness amplification of weakly verifiable puzzles. In: TCC. pp. 17–33 (2005)
28. Charles Lee: Litecoin (2011), https://litecoin.info/
29. Chen, L., Morrissey, P., Smart, N.P., Warinschi, B.: Security notions and generic constructions for client puzzles. In: ASIACRYPT. LNCS, vol. 5912, pp. 505–523 (2009)

30. Chepurnoy, A., Duong, T., Fan, L., Zhou, H.S.: Twinscoin: A cryptocurrency via proof-of-work and proof-of-stake. Cryptology ePrint Archive, Report 2017/232 (2017), http://eprint.iacr.org/2017/232
31. Cook, S.A.: An observation on time-storage trade off. In: STOC. pp. 29–33 (1973)
32. Cox, B.: Twocats (and skinnycat): A compute time and sequential memory hard password hashing scheme. Password Hashing Competition. v0 edn. (2014)
33. Dean, D., Stubblefield, A.: Using client puzzles to protect TLS. In: USENIX Security. vol. 10 (2001)
34. Demay, G., Gaži, P., Hirt, M., Maurer, U.: Resource-restricted indifferentiability. In: EUROCRYPT. LNCS (2013)
35. Demay, G., Gaži, P., Maurer, U., Tackmann, B.: Query-complexity amplification for random oracles. In: ICITS. LNCS (2015)
36. Dodis, Y., Guo, S., Katz, J.: Fixing cracks in the concrete: Random oracles with auxiliary input, revisited. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology — EUROCRYPT 2017. LNCS, vol. 10211, pp. 473–495. Springer (2017)
37. Dwork, C., Goldberg, A., Naor, M.: On memory-bound functions for fighting spam. In: CRYPTO. LNCS, vol. 2729, pp. 426–444 (2003)
38. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: CRYPTO. pp. 139–147 (1992)
39. Dwork, C., Naor, M., Wee, H.: Pebbling and proofs of work. In: CRYPTO. LNCS, vol. 3621, pp. 37–54 (2005)
40. Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: CRYPTO. pp. 585–605. LNCS (2015)
41. Dziembowski, S., Kazana, T., Wichs, D.: Key-evolution schemes resilient to space-bounded leakage. In: CRYPTO. LNCS, vol. 6841, pp. 335–353 (2011)
42. Dziembowski, S., Kazana, T., Wichs, D.: One-time computable self-erasing functions. In: TCC. LNCS, vol. 6597, pp. 125–143 (2011)
43. Eckey, L., Faust, S., Loss, J.: Efficient algorithms for broadcast and consensus based on proofs of work. Cryptology ePrint Archive, Report 2017/915 (2017), http://eprint.iacr.org/2017/915
44. Forler, C., Lucks, S., Wenzel, J.: Catena: A memory-consuming password scrambler. Cryptology ePrint Archive, Report 2013/525 (2013)
45. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol. In: EUROCRYPT. LNCS (2015)
46. Garay, J.A., Kiayias, A., Panagiotakos, G.: Proofs of work for blockchain protocols. Cryptology ePrint Archive, Report 2017/775 (2017), http://eprint.iacr.org/2017/775
47. Garay, J.A., MacKenzie, P., Prabhakaran, M., Yang, K.: Resource fairness and composability of cryptographic protocols. In: TCC. LNCS, vol. 3876, pp. 404–428 (2006)
48. Groza, B., Petrica, D.: On chained cryptographic puzzles. In: SACI. pp. 25–26 (2006)
49. Groza, B., Warinschi, B.: Cryptographic puzzles and DoS resilience, revisited. DCC 73(1), 177–207 (2014)
50. Hewitt, C.E., Paterson, M.S.: Record of the project mac. In: Conference on Concurrent Systems and Parallel Computation. pp. 119–127. ACM, New York, NY, USA (1970)
51. Jeckmans, A.: Practical client puzzle from repeated squaring (August 2009)
52. Jerschow, Y.I., Mauve, M.: Non-parallelizable and non-interactive client puzzles from modular square roots. In: ARES. pp. 135–142. IEEE (2011)

53. Jr., M.A.S., Almeida, L.C., Andrade, E.R., dos Santos, P.C.F., Barreto, P.S.L.M.: Lyra2: Password Hashing Scheme with improved security against time-memory trade-offs
54. Juels, A., Brainard, J.G.: Client puzzles: A cryptographic countermeasure against connection depletion attacks. In: NDSS (1999)
55. Karame, G.O., Čapkun, S.: Low-cost client puzzles based on modular exponentiation. In: ESORICS. pp. 679–697 (2010)
56. Lengauer, T., Tarjan, R.E.: Asymptotically tight bounds on time-space trade-offs in a pebble game. J. ACM 29(4), 1087–1130 (Oct 1982)
57. Mahmoody, M., Moran, T., Vadhan, S.: Publicly verifiable proofs of sequential work. In: ITCS. pp. 373–388. ITCS '13 (2013)
58. Mahmoody, M., Moran, T., Vadhan, S.P.: Time-lock puzzles in the random oracle model. In: CRYPTO. LNCS, vol. 6841, pp. 39–50 (2011)
59. Markus, B.: Dogecoin (2013), `http://dogecoin.com/`
60. Maurer, U.: Indistinguishability of random systems. In: EUROCRYPT. LNCS, vol. 2332, pp. 110–132 (2002)
61. Maurer, U.: Constructive cryptography: A new paradigm for security definitions and proofs. In: TOSCA. LNCS (2011)
62. Maurer, U., Renner, R.: Abstract cryptography. In: ICS (2011)
63. Maurer, U., Renner, R.: From indifferentiability to constructive cryptography (and back) from indifferentiability to constructive cryptography (and back). In: TCC. LNCS, vol. 9985, pp. 3–24 (2016)
64. Maurer, U., Renner, R., Holenstein, C.: Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In: TCC. LNCS, vol. 2951, pp. 21–39 (2004)
65. Morris, R., Thompson, K.: Password security: A case history. Commun. ACM 22(11), 594–597 (November 1979)
66. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2009)
67. Naor, M.: Moderately hard functions: From complexity to spam fighting. In: FST TCS 2003. LNCS, vol. 2914, pp. 434–442 (2003)
68. Percival, C.: Stronger key derivation via sequential memory-hard functions. In: BSDCan 2009 (2009)
69. Pfitzmann, B., Waidner, M.: A model for asynchronous reactive systems and its application to secure message transmission. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy. pp. 184–200. IEEE (2001)
70. Pintér, K.: Gambit – A sponge based, memory hard key derivation function. Submission to Password Hashing Competition (PHC) (2014)
71. Price, G.: A general attack model on hash-based client puzzles. In: Cryptography and Coding. LNCS, vol. 2898, pp. 319–331 (2003)
72. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with composition: Limitations of the indifferentiability framework. In: Paterson, K.G. (ed.) Advances in Cryptology — EUROCRYPT 2011. vol. 6632, pp. 487–506. Springer (2011)
73. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep., Cambridge, MA, USA (1996)
74. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep., Cambridge, MA, USA (1996)
75. Stebila, D., Kuppusamy, L., Rangasamy, J., Boyd, C., González Nieto, J.: Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols. In: CT-RSA. LNCS, vol. 6558, pp. 284–301 (2011)
76. Stebila, D., Ustaoglu, B.: Towards denial-of-service-resilient key agreement protocols. In: (ACISP) 2009. LNCS, vol. 5594, pp. 389–406 (2009)

77. Tang, Q., Jeckmans, A.: On non-parallelizable deterministic client puzzle scheme with batch verification modes (January 2010), http://doc.utwente.nl/69557/
78. Thompson, C.D.: Area-time complexity for vlsi. In: Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing. pp. 81–88. STOC '79, ACM, New York, NY, USA (1979), http://doi.acm.org/10.1145/800135.804401
79. Tritilanunt, S., Boyd, C.A., Foo, E., Nieto, J.M.G.: Toward non-parallelizable client puzzles. In: Cryptology and Network Security. LNCS, vol. 4856, pp. 247–264 (2007)
80. Unruh, D.: Random oracles and auxiliary input. In: Menezes, A. (ed.) Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4622, pp. 205–223. Springer (2007), http://dx.doi.org/10.1007/978-3-540-74143-5_12
81. Wu, H.: POMELO – A Password Hashing Algorithm (2015)
82. Yao, F.F., Yin, Y.L.: Design and analysis of password-based key derivation functions. In: Menezes, A. (ed.) Topics in Cryptology — CT-RSA 2005. LNCS, vol. 3376, pp. 245–261. Springer (2005)

A The embedding lemma

We aim at using the standard composition theorem of constructive cryptography [62,61] to compose the MoHFs proven secure according to Definition 5 with the application protocols described in Sections 7 and 8. We note, however, that the indifferentiability statement according to Definition 5 is not immediately applicable in the case with two honest parties, as required in the availability conditions of Definitions 10 and 11, where both the prover and verifier are honest. In particular, the priv-interface models use by an honest party, whereas the pub-interface corresponds to a dishonest party. We argue here, however, that for stateless algorithms a definition with two honest users' interfaces priv_1 and priv_2 follows immediately: as potential oracles are evaluated consistently at the interfaces priv_1 and priv_2 and the algorithm is stateless, there is no difference between providing input at those two interfaces, as long as the resources at the single interface priv are sufficient to make the same number of evaluations. This statement is formalized in the following lemma, where the resources $\mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}}$ and $\mathcal{T}^{\text{RRO}}_{a_1,a_2,b}$ are defined analogously to $\mathcal{S}_{\mathbf{l},\mathbf{r}}$ and $\mathcal{T}^{\text{RRO}}_{a,b}$ but with two "honest" interfaces priv_1 and priv_2.

Lemma 1. *Let* $f^{(\cdot)}, \mathtt{naïve}, \mathbb{P}$, $\mathsf{a},\mathsf{b} : \mathbb{P} \to \mathbb{N}$, *and* $\varepsilon : \mathbb{P} \times \mathbb{P} \to \mathbb{R}_{\geq 0}$ *as in Definition 5, with* $\mathtt{naïve}$ *being a stateless algorithm, and* $\mathsf{a}(\mathbf{l}_1 + \mathbf{l}_2) \geq \mathsf{a}(\mathbf{l}_1) + \mathsf{a}(\mathbf{l}_2)$ *for all* $\mathbf{l}_1, \mathbf{l}_2 \in \mathbb{P}$.

For a family of models $\mathcal{S}_{\mathbf{l},\mathbf{r}}$, *let* $(f^{(\cdot)}, \mathtt{naïve})$ *be a* $(\mathsf{a},\mathsf{b},\varepsilon)$*-secure moderately hard uniform function in the* $\mathcal{S}_{\mathbf{l},\mathbf{r}}$*-model. Let* $\mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}}$ *and* $\mathcal{T}^{\text{RRO}}_{a_1,a_2,b}$ *be as described above. Then,*

$$\forall \mathbf{r} \in \mathcal{P}\ \exists \sigma\ \forall \mathbf{l}_1, \mathbf{l}_2 \in \mathcal{P} : \quad \pi^{\text{priv}_1}\ \pi^{\text{priv}_2}\ \mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}} \quad \approx_{\varepsilon(\mathbf{l}_1+\mathbf{l}_2,\mathbf{r})} \quad \sigma^{\text{pub}}\ \mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}_1),\mathsf{a}(\mathbf{l}_2),\mathsf{b}(\mathbf{r})}\,, \tag{1}$$

where $\pi = \pi_{\mathtt{naïve}}$ *is the protocol from Definition 5.*

Proof. For simplicity of notation, we define

$$\mathcal{S}' \coloneqq \pi^{\text{priv}_1}\ \pi^{\text{priv}_2}\ \mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}}\ , \qquad \mathcal{T}' \coloneqq \sigma^{\text{pub}}\ \mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}_1),\mathsf{a}(\mathbf{l}_2),\mathsf{b}(\mathbf{r})}\ ,$$
$$\mathcal{S}'' \coloneqq \pi^{\text{priv}}\ \mathcal{S}_{\mathbf{l}_1+\mathbf{l}_2,\mathbf{r}}\ , \qquad \mathcal{T}'' \coloneqq \sigma^{\text{pub}}\ \mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}_1+\mathbf{l}_2),\mathsf{b}(\mathbf{r})}\ .$$

We prove the statement by reduction. Let σ be the simulator that exists by the validity of Definition 5; the same σ is used in equation (1). Let D be a distinguisher that achieves advantage $\varepsilon' > \varepsilon(\mathbf{l}_1 + \mathbf{l}_2, \mathbf{r})$; we build a distinguisher $D'(D)$ for

$$\mathcal{S}'' = \pi^{\text{priv}}\ \mathcal{S}_{\mathbf{l}_1+\mathbf{l}_2,\mathbf{r}} \quad \approx_{\varepsilon(\mathbf{l}_1+\mathbf{l}_2,\mathbf{r})} \quad \sigma^{\text{pub}}\ \mathcal{T}^{\text{RRO}}_{\mathsf{a}(\mathbf{l}_1+\mathbf{l}_2),\mathsf{b}(\mathbf{r})} = \mathcal{T}''$$

as follows. All queries that D makes to interfaces priv_1 or priv_2 are instead made at the single interface priv; up to $\mathsf{a}(\mathbf{l}_1)$ queries at priv_1 and up to $\mathsf{a}(\mathbf{l}_2)$ queries at priv_2, all further queries are answered with $\perp$ instead. The queries at pub are simply forwarded to the respective interface of $\mathcal{S}'$ or $\mathcal{T}'$, respectively.

By the definition of $\mathcal{S}_{\mathbf{l}_1,\mathbf{l}_2,\mathbf{r}}$ and $\mathcal{T}^{\text{RRO}}_{a_1,a_2,b}$, the fact that $\mathtt{naïve}$ is stateless, and $\mathsf{a}(\mathbf{l}_1 + \mathbf{l}_2) \geq \mathsf{a}(\mathbf{l}_1) + \mathsf{a}(\mathbf{l}_2)$, the views of D when connected to $\mathcal{S}'$ or $D'(\cdot)\mathcal{S}''$, and respectively to $\mathcal{T}'$ or $D'(\cdot)\mathcal{T}''$, are the same. □

B Proof of Theorem 4

To prove the two lemmas, we use a standard compression argument to show that violating them requires compressing the function table of a RO with large probability; an impossible task. For this the proof uses the following lemma bounding the success probability of a predictor at guessing bit-strings when given short but correlated hints. (A proof can be found in [42] for example.)

Lemma 2. *Let $B = b_1, \ldots, b_u$ be a sequence of uniform random bits. Let $\mathtt{P}$ be a randomized procedure which gets a hint $h \in \Gamma$, and can adaptively query any of the bits of B by submitting an index i and receiving b_i. At the end of the execution $\mathtt{P}$ outputs a subset $S \subseteq \{1, \ldots, u\}$ of $|S| = k$ indices which were not previously queried, along with guesses for all of the bits $\{b_i | i \in S\}$. Then the probability (over the choice of B and randomness of $\mathtt{P}$) that there exits some $h \in \Gamma$ for which $\mathtt{P}(h)$ outputs* all correct guesses *is at most $\frac{|\Gamma|}{2^k}$.*

The simulator. We begin with the detailed description of the simulator σ. Recall (from Definitions 5 and 6) that σ is connected to the pub-interface of $\mathcal{T}^{\text{RRO}}_{l,r}$ where $l = \mathsf{a}_w(q_l, m_l)$ and $r = \mathsf{b}_w(q_r, m_r)$. Upon initialization, σ initializes an internal copy of $\mathcal{S}^{w\text{-PROM}}_{\mathbf{l},\mathbf{r}}$ where $\mathbf{l} = (q_l, m_l)$ and $\mathbf{r} = (q_r, m_r)$. To be precise, σ implements h using a lazy-sampling technique: Whenever h is queried, σ checks whether that query has been made before. If so, σ responds with the same answer. Otherwise it selects a uniform random $y \leftarrow\!\!\$\ \{0,1\}^w$, records the pair (x, y) and responds with y. Once initialization is complete, upon input of a pROM algorithm $\mathtt{A}$ and input x_{in} on its external interface connected with distinguisher D, the simulator provides $\mathtt{A}$ and x_{in} to the emulated copy of $\mathcal{S}^{w\text{-PROM}}$. This causes $\mathcal{S}^{w\text{-PROM}}$ to

(sample fresh random coins \$ and) begin the execution of $\mathtt{A}(x_{\mathsf{in}}; \$)$. During the emulation of that execution, σ keeps track of the calls to h made by $\mathtt{A}$. All calls are answered using the lazy sampling method unless $\mathtt{A}$ is making a call of the form $h(\bar{x})$ with $\bar{x} = (x, u, \ell_1, \ell_2)$ where all of the following three conditions are met:

1. $u = v_{\mathsf{out}}$ is the sink of G,
2. ℓ_1 and ℓ_2 are the labels of the parents of u in G in the (h, x)-labeling of G,
3. $\mathtt{A}$ has already made all other calls to h for the (h, x)-labeling of G in an order respecting the topological sorting of G.

We call a query to h, for which the first two conditions are valid, an *h-final call (for x)*. If the third condition also holds then we call the query a *sound* final call. For a sound final call $\bar{x}$, if the record $(\bar{x}, y)$ already exists, then respond with y. Otherwise, send x to $\mathcal{T}^{\text{RRO}}$ (via the pub interface). If the response is $y \in \{0,1\}^w$, then store the record $(\bar{x}, y)$ and continue emulating $\mathtt{A}$ with y as the response to $h(\bar{x})$. If, however, the response from $\mathcal{T}^{\text{RRO}}$ is $\perp$, then σ outputs $\perp$ to D and halts. Whenever $\mathcal{S}^{w\text{-PROM}}$ produces output on its pub-interface, σ forwards that output on to D and halts (ignoring all future messages).

It remains to be proved that Definition 5 is satisfied, that there is no distinguisher D that can tell apart an interaction with $\pi^{\text{priv}}_{\mathtt{naïve}_G}\ \mathcal{S}^{w\text{-PROM}}_{\mathbf{l},\mathbf{r}}$ from an interaction with $\sigma^{\text{pub}}\ \mathcal{T}^{\text{RRO}}_{l,r}$ with probability greater than $\epsilon_w(\mathbf{l}, \mathbf{r}, n)$. To this end, we introduce the following generalization of the pebbling game which serves to model concurrent evaluations of f_G on multiple inputs.

Multi-color pebbling. Intuitively, the pebbling game above abstracts a single computation of a graph function. Placing a pebble on a node corresponds to computing a label and storing its value for future use. However, because we are ultimately concerned with the *amortized* complexity of MoHFs, we must consider repeated computations of functions in F. Therefore, we introduce the following multi-colored generalization of the pebbling game. Intuitively each color represents the values stored while evaluating a distinct (index, input) pair.

More precisely, let G be a DAG and $m \in \mathbb{N}$. The m-tuple $P = (P^1, \ldots, P^m)$ is called an *m-color pebbling* of G (or just m-pebbling for short) if each P^j is a complete and legal pebbling of G. Moreover, if P is an m-pebbling of G and $\mathit{\Pi}$ is the set of all m-pebblings of G then the *m-cumulative pebbling complexity* (m-CPC) of G is defined to be:

$$\mathsf{cpc}_m(P) := \sum_{j \in [m]} \sum_i \left|P^i_j\right|, \qquad \mathsf{cpc}_m(G) := \min\left\{\mathsf{cpc}_m(P) : P \in \mathit{\Pi}\right\}.$$

It is immediate from this definition that cpc_m of any graph scales linearly in the number of colors.

Lemma 3. *For any DAG G and positive integer $m > 0$ it holds that:*

$$\mathsf{cpc}_m(G) = m \cdot \mathsf{cpc}(G).$$

Ex-post-facto pebbling. We define (a variant of) the ex-post-facto m-pebbling P (which were first introduced in [39]) of a given execution E in the pROM. We prove two important properties relating P to E. First, w.h.p., if E contains m final calls then P is a legal and complete m-pebbling of G. Second, w.h.p., if for the bit-length of the input state $|\sigma_i| = d$, then $\sum_{j\in[m]} \left|P_i^j\right|$ is at most of size (approximately) d/w. Thus, w.h.p., both statements are true for P and so we can lower-bound $\mathsf{cmc}(E)$ in terms of $\sum_i \sum_{j\in[m]} \left|P_i^j\right|$ which is at least $m \cdot \mathsf{cpc}(G)$.

The following is a generalization of the ex-post-facto pebbling in [11,42] such that every distinct index/input pair appearing in a RO call is assigned its own color in the m-coloring.

Fix an execution of some algorithm $\mathtt{A}$ in the pROM (i.e., as executed by $\mathcal{S}^{w\text{-PROM}}$). More precisely, fix an execution of algorithm $\mathtt{A}$ on input X using coins \$ with oracle h resulting in $z \in \mathbb{N}$ output states $\bar{\sigma}_1, \ldots, \bar{\sigma}_z$ where $\bar{\sigma}_i = (\tau_i, \mathbf{q}_i)$ and $\mathbf{q}_i = (q_i^1, q_i^2, \ldots, q_i^{y_i})$. To define the corresponding *ex-post-facto* m-color pebbling of G' we need the following terminology.

For input $x \in \{0,1\}^*$ and node $v \in V$ let $\mathsf{pre\text{-}lab}(x, v)$ denote the RO query $(x, v, \mathsf{lab}(v_1), \ldots, \mathsf{lab}(v_z))$, where the $\{v_j\}$ are the parents of v arranged in lexicographic order and $\mathsf{lab}(v_i)$ is the label of v_i in the (h, x)-labeling of G.[32] A RO query $q \in \{0,1\}^*$ is said to be *correct for* (x, v) if $q = \mathsf{pre\text{-}lab}(x, v)$. In this case the nodes $\mathsf{parents}(v) \subseteq V$ are called the *input nodes* of q and $v \in V$ is the *output node* of q.

We can now define the ex-post-facto pebbling for the execution. Initially, all sets P_i^x for $i \in [0, z]$ and $x \in \{0,1\}^*$ are empty. We start with $i = 1$ and parse the batches of RO calls in the output states of the execution in the order they were produced, populating the sets as we go along by applying the following rules to each batch $\mathbf{q}_i$.

1. Initially, for all $x \in \{0,1\}^*$, we set $P_i^x := P_{j-1}^{i,x}$.
2. For each query $q \in \mathbf{q}_i$, if it is correct for some $(x, v) \in \{0,1\}^* \times V$, then set $P_i^x := P_i^x \cup \{v\}$.[33]
3. For fixed input $x \in \{0,1\}^*$, a node $v \in V$ is called *necessary for* x if there exists a future query $q \in \mathbf{q}_l$ with $l > j$ where v is an input node to a correct call q for some (x, u), yet in between there is no correct call for (x, v). That is there is no $q' \in \mathbf{q}_s$ with $j < s < l$ where q' is a correct call (x, u). Remove all v from P_j^x which are *not* necessary.[34]

[32] In other words $\mathsf{pre\text{-}lab}(x, v)$ is precisely the RO call defining the label of v in the (h, x)-labeling of G.

[33] Intuitively, placing a pebble on v (by adding v to set P_j^x) represents $\mathtt{A}$ storing in memory v's label for the (h, x) labeling of G.

[34] Intuitively, v is necessary for x if the label of v will be needed to make some future call to h in the computation of the (h, x) labeling of G, but the label of v in that labeling will not be recomputed beforehand. Only necessary nodes need to be stored (i.e. pebbled) since other nodes are either not needed, or will be computed (i.e. pebbled) before they are needed. We optimistically assume that the labels of unnecessary nodes are never stored.

Finally let $X \subseteq \{0,1\}^*$ be the set of inputs to f_G such that $\exists i \in [z]$ such that $v_{\mathsf{out}} \in P_i^x$. Then the ex-post-facto pebbling of this execution is the $|X|$-color pebbling $P := \{P^x = (P_1^x, \ldots, P_z^x) \mid x \in X\}$.

We prove that, w.h.p., P is indeed a multi-pebbling of G (i.e. for each color in P the pebbling is both complete and legal for G).

Claim. For any input x_{in}, the probability that the ex-post-facto pebbling P of an execution $\mathtt{A}^h(x_{\mathsf{in}}; \$)$ is a legal and complete $|X|$-color pebbling of G is at least $1 - \frac{q_r}{2^w}$ over the choice of h and \$.

Proof. We must show that P is both legal and complete with high probability.

To see that P must always be complete notice that, by construction, $\forall x \in X$ it holds that $P_0^x = \emptyset$. Moreover by definition of X, for all $P^x \in P$, there exists some time step where v_{out} is pebbled in P^x. Thus each P^x is a complete pebbling of G and thus so is P.

To prove that P is legal we build a predictor $\mathtt{P}$ which attempts to compress parts of the RO function table by making internal use of $\mathtt{A}$. We show that for any combination of random coins \$ and RO h that causes an execution for which the ex-post-facto pebbling P is illegal, $\mathtt{P}$ will succeed at compressing parts of h when given \$. Lemma 2 then implies that such pairs $(\$, h)$ must be very rare.

We call a RO call by $\mathtt{A}$ *lucky* if it is correct for some (x, v), has input node w but no previous correct call for (x, w) was made. Notice that the only way for P to be illegal is if $\mathtt{A}$ makes such a lucky call. Our goal is to build a predictor $\mathtt{P}$ as in Lemma 2 which is given access to the random string B. The predictor $\mathtt{P}$ depends on x_{in}, the value supplied by D as input to the execution of $\mathtt{A}$. It uses the even bits of B as the random coins \$ and odd bits of B as a function table of h to internally emulate a copy of $\mathtt{A}^h(x_{\mathsf{in}}; \$)$ as follows.

Hint: Recall that $\mathtt{A}$ is permitted by $\mathcal{S}^{w\text{-PROM}}$ to make at most q_r RO calls in total. The predictor receives as a hint the index $j \in [q_r]$ of the (first) lucky RO made by $\mathtt{A}$. If no lucky call exists for that choice of B (i.e. the \$ and h induced by B) then no hint is given and $\mathtt{P}$ simply halts.

Execution: The predictor runs $\mathtt{A}^h(x; \$)$ forwarding all RO calls to h (i.e. by reading B at the relevant positions) until the j^{th} query q. By assumption q is lucky which means it is correct for some (x, v) and contains the label $\mathsf{lab}(w)$ of w in the (h, x)-labeling of G although no correct call for (x, w) was previously made. The reduction recomputes the value of $\mathsf{pre\text{-}lab}(x, w)$ by evaluating the (h, x)-labeling in topological order. Notice this is done without querying $\mathcal{H}(\mathsf{pre\text{-}lab}(x, w))$. Finally it outputs $\mathsf{lab}(w)$ as its guess for the odd bits of B at the positions representing $\mathsf{pre\text{-}lab}(x, w)$.

Notice that $\mathtt{P}$ outputs a correct guess whenever $\mathtt{A}$ produces a lucky RO call. Moreover $\mathtt{P}$ can receive a total of q_r different values for the hint nor does $\mathtt{P}$ ever read out the the w positions (representing) $\mathsf{lab}(w)$ in the string B. Thus Lemma 2 implies that $\mathtt{P}$ can succeed with probability at most $q_r/2^w$, which in turn implies that $\mathtt{A}$ can produce a lucky RO call with that probability. This concludes the proof of the claim. □

Now we show that at the end of any step in the execution, w.h.p., the sum of the number of pebbles (over all colors) on G can be upper-bounded by the size of the data τ_i in the input state σ_i.

Claim. Fix any input x_{in}. Let σ_i be the i^{th} input state in an execution of $\mathtt{A}^h(x_{\mathsf{in}};\$)$. Then, for all $\lambda \geq 0$,

$$\Pr\left[\forall i \ : \ \sum_{x \in X} |P_i^x| \leq \frac{|\sigma_i| + \lambda}{w - \log(q_r)}\right] > 1 - 2^{-\lambda}$$

over the choice of h and \$.

Proof. Recall that, intuitively, we only place necessary pebbles in P, namely ones representing labels which will be used in a future call without first being recomputed. Suppose that, at some time step $i \in [z]$, P has a large number of pebbles on G but $\mathtt{A}$ was only given a small state σ_i. Then, intuitively, we should be able to use $\mathtt{A}$ to predict the values of the labels of the pebbled nodes given only a small hint (namely σ_i). Since random oracles are w.h.p. incompressible this event should happen with low probability. We make this intuition formal with the following reduction $\mathtt{P}$ whose goal it is to predict the values of h at the points $\mathsf{pre\text{-}lab}(x, v)$ for all pebbled nodes in such a time step.

Fix any input x_{in} and let $\sigma_0, \sigma_1, \ldots \sigma_z$ be the input states for the execution $\mathtt{A}^h(x_{\mathsf{in}};\$)$. Let P be the corresponding ex-post-facto pebbling. Fix any step $i \in [z]$. The reductions goal will be to predict the label of all pebbles (of any color) currently on G in the i^{th} step of P.

A query q to h made by $\mathtt{A}$ at time $j > i$ is called *critical for* (x, v) if q is correct for some node (x, u) and v is an input-node for q but no correct call for v was made during the time steps $[i + 1, j - 1]$. Intuitively, critical queries are interesting because, by telling the predictor when they will occur (with a hint), the predictor can use them to extract the labels of all nodes that are necessary at time i. Note that these are exactly the set of pebbles on G at time i. In particular, there can be at most $c \leq \sum_{x \in X} |P_i^x|$ critical calls since there can be at most one per pebble on G.

As before, $\mathtt{P}$ has access to a bit-string B which it uses to implement coins \$ and a RO h in an emulation of $\mathtt{A}^h(x_{\mathsf{in}};\$)$. The prediction proceeds as follows.

Hint: As a hint it is given $\mathcal{I} \in [q]^c$, the indices of the critical calls made by $\mathtt{A}$, and the state σ_i.

Execution: The reduction runs $\mathtt{A}$ with initial input state σ_i. When $\mathtt{P}$ gets an oracle call q, it checks if q is critical (in some (h, x)-labeling) using $\mathcal{I}$. If so, then, for each input node v of q, the reduction records the corresponding label in q as v's label in the (h, x)-labeling of G. These will eventually become the predictions for the RO. To answer q, the reduction does the following.

- Determines if the call is correct for some (x, v). This is the case if q is critical. It is also correct for (x, v) if q has the form $q = (x, v, \ell_1, \ldots, \ell_e)$, node v has e parents in G, and, for each parent v_l, a correct call for (i, x, v_l) has already been made and that call's result was ℓ_l. In particular,

P can determine if $q = \mathsf{pre\text{-}lab}(x, v)$ and no new calls need be made by P to check this.

– If the call is correct for (x, v) and the label $\mathsf{lab}(v)$ in the (h, x)-labeling has been recorded, then respond to A with this label. Otherwise query $h(q)$, record the answer, and return it to A.

When A terminates, P has recorded all labels of input-nodes to all critical calls. These will serve as its predictions for the outputs of the RO. So what remains is computing the corresponding inputs as follows.

First P checks the transcript to determine the sets $\{P_i^x \mid x \in X\}$.[35] For all $x \in X$ and $v \in P_i^x$, the reduction computes $\mathsf{pre\text{-}lab}(x, v)$ using h and the recorded labels extracted from critical calls. Finally, it outputs $\mathsf{lab}(x, v)$ as its prediction of $h(\mathsf{pre\text{-}lab}(x, v))$.

Notice that the labels in the output of P were all used in a critical call without first being recomputed by A. Thus, P never queried them to h, as they were recorded before $\mathsf{pre\text{-}lab}(x, v)$ was ever queried by A. It follows that P correctly predicts the value of h at $\mathsf{pre\text{-}lab}(x, v)$, without querying h at that point, for every pair $(x, v) \in X \times V$ with $v \in P_i^x$.

Now assume, for the sake of contradiction, that $\exists \lambda \geq 0$ such that with probability at least $2^{-\lambda}$ for some $i \in [z]$ it holds that if we define $\mu := \sum_x |P_i^x|$, then $\mu > \frac{|\sigma_i| + \lambda}{w - \log(q)}$. Let $\mathcal{E}$ be the event that the reduction produces correct predictions for all labels it attempts. By construction $\Pr[\mathcal{E}] \geq 2^{-\lambda}$. On the other hand, using Lemma 2 and the fact that $c \leq \sum_x |P_i^x| = \mu$, we can now write:

$$\Pr[\mathcal{E}] \leq \frac{q^r \cdot 2^{|\sigma_i|}}{2^{\mu w}} \leq 2^{\mu(\log(q) - w) + |\sigma_i|} .$$

But this implies $2^{-\lambda} \leq 2^{\mu(\log(q) - w) + |\sigma_i|}$, which is a contradiction to the assumption. In other words, there can be no such λ and so we are done. □

We are now equipped to complete the proof of Theorem 4.

Fix any $\lambda \geq 0$, $z \in \mathbb{N}$ and define

$$\rho := q/2^w + 2^{-\lambda} .$$

Next, fix $\mathbf{l}, \mathbf{r} \in \mathbb{P}^{\text{PROM}}$ and set $l = \mathsf{a}(\mathbf{l})$ and $r = \mathsf{b}(\mathbf{r})$, our goal is to prove that

$$\pi_{\mathtt{naïve}_G}{}^{\text{priv}} \; \mathcal{S}_{\mathbf{l},\mathbf{r}}^{w\text{-PROM}} \quad \approx_\rho \quad \sigma^{\text{pub}} \; \mathcal{T}_{l,r}^{\text{RRO}} .$$

Let D be any distinguisher. By definition of the real and ideal model, the interaction proceeds as follows: First D makes (some number of) queries at the priv-interface. Then it inputs an algorithm at the pub-interface, and obtains the outputs. Finally, it again makes (some number of) queries at the priv-interface, before it outputs the decision bit. We discuss each of these phases individually.

[35] This can be done by adding v to P_i^x if and only if v is an input node to some future critical call for the (h, x)-labeling.

For the first phase, the outputs of $\mathcal{I} \coloneqq \sigma^{\text{pub}}\ \mathcal{T}_{l,r}^{\text{RRO}}$ for distinct inputs at the priv-interface are uniformly random and independent by the definition of $\mathcal{T}_{l,r}^{\text{RRO}}$. In the case of $\mathcal{R} \coloneqq \pi_{\texttt{naïve}_G}{}^{\text{priv}}\ \mathcal{S}_{\mathbf{l},\mathbf{r}}^{w\text{-PROM}}$, proper prefixing the construction of the graph function assures that all final queries the function makes to $\mathcal{S}_{\mathbf{l},\mathbf{r}}^{w\text{-PROM}}$ are on fresh inputs, so the outputs are also uniformly random and independent. Therefore, the systems behave equivalently up to this point.

For the second phase, we first consider a variation $\hat{\mathcal{I}}$ of $\mathcal{I}$ in which the simulator σ is allowed to issue more than r queries to make the simulation valid. We, then, define events bad that occur in the execution of $\mathtt{A}$ within $\hat{\mathcal{I}}$ and $\mathcal{R}$ whenever $\mathtt{A}$ makes a lucky call to the RO. The evaluations of $\hat{\mathcal{I}}$ and $\mathcal{R}$, conditioned on $\neg\mathsf{bad}$ and on the prior inputs/outputs from the queries at priv, are identical: for any (fresh) non-final query, or final query that is not determined by the interaction at the priv-interface, the response is uniformly random. For a final query that is determined by the prior interaction, it is determined. Using Maurer's conditional-equivalence theorem [60], we obtain that the distinguishing advantage between $\hat{\mathcal{I}}$ and $\mathcal{R}$ up to this point is bounded by the probability of bad, and by the first above claim we can bound this by $q/2^w$. The distinguishing advantage between $\hat{\mathcal{I}}$ and $\mathcal{I}$ is bounded through the second claim by $2^{-\lambda}$.

For the third phase, we argue analogously to the first phase. (Final) queries that have been answered before, either at the priv-interface or toward $\mathtt{A}$, are answered consistently in both $\mathcal{I}$ and $\mathcal{R}$, and the responses to all other queries are uniformly random. Overall, we conclude that the distinguishing advantage can be bounded by ρ, which concludes the proof.

C niPoE with honest and dishonest provers

The niPoE material in Section 8 considers only settings where all parties are honest or all parties are dishonest. In this section, we consider the more complex but realistic setting where both honest and dishonest provers exist.

C.1 Definition

The main difference between the material in this section and the one in Section 8 is that all resources have three interfaces: an interface P for the honest prover, an interface V for the verifier, and a third interface E for the attacker.

The goal of niPoE protocols. The constructed resource is similar to the resource NIPOE described in Section 7.1, with the main difference being that honest and dishonest provers appear in the same setting. Consequently, the resource NIPOE takes as input at both the P- and E-interfaces statements $s \in \mathcal{S}$, and returns 1 if the proof succeeded and 0 otherwise. The E-interface additionally allows to retrieve the list of statements for which a proof has been performed at the P-interface. Upon an activation at the verifier's interface V, if for any statement $s \in \mathcal{S}$ a proof has been successful, the resource outputs this s, and it outputs $\perp$ otherwise. An output $s \neq \perp$ has the meaning that the party at the P- or

E-interface has spent enough effort for the particular statement s. Similarly to NIPOE, the resource NIPOE is parametrized by a performance function $\phi : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and the number $\underline{b} \in \mathbb{N}$ of verification attempts. The provers' attempts are specified by a number $\underline{a} \in \mathbb{N}$ of attempts for the honest prover and a number $\overline{a} \in \mathbb{N}$ for the dishonest prover. The resource is denoted as $\mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,\underline{b}}$. The behavior of this resource is described in more detail in Figure 6. There are three inputs at E that need further explanation:

- (copy, s): This corresponds to sending a proof to V. Prover V is convinced if the proof was successful (i.e., $\hat{e}_s = 1$), and has to spend one additional evaluation of $\mathcal{T}^{\mathrm{RRO}}$, so the corresponding counter is increased ($d \leftarrow d + 1$).
- (spend): E forces V to spend one additional evaluation of $\mathcal{T}^{\mathrm{RRO}}$, for instance by sending an invalid proof. This decreases the number of verifications that V can still do ($d \leftarrow d + 1$).
- (read): E may observe the proofs that P has sent to V.

Non-interactive proof-of-effort resource, 3-party setting

The resource is parametrized by numbers $\underline{a}, \overline{a}, \underline{b} \in \mathbb{N}$ and a mapping $\phi : \mathbb{N} \to \mathbb{R}_{\geq 0}$. It contains as state bits $\hat{e}_s \in \{0,1\}$ and counters d as well as $c_s, \hat{c}_s \in \mathbb{N}$ for each $s \in \{0,1\}^*$ (all initially 0), and lists $\mathsf{S}, \hat{\mathsf{S}} \in (\{0,1\}^*)^*$ of strings that is initially empty.

Verifier V: On input a unary value, if S is empty then return $\perp$. Otherwise remove the first element of S and return it.

Honest prover P: On input a string $s \in \{0,1\}^*$, set $c_s \leftarrow c_s + 1$. If $\sum_{s \in \{0,1\}^*} c_s > a$, then return 0. Otherwise, draw $e \in \{0,1\}$ at random such that it is 1 with probability $\phi(c_s)$ and 0 otherwise. If $e = 1$ then append s to $\hat{\mathsf{S}}$. If $d < b$, then append s to S and set $d \leftarrow d + 1$. Output e at interface P.

Dishonest prover P:
- On input a string $s \in \{0,1\}^*$, set $c_s \leftarrow c_s + 1$. If $\sum_{s \in \{0,1\}^*} c_s > a$, then return 0. Otherwise, of $\hat{e}_s = 0$, then draw $\hat{e}_s$ at random such that it is 1 with probability $\phi(c_s)$ and 0 otherwise. Output $\hat{e}_s$ at interface P.
- Upon an input (copy, s). Set $d \leftarrow d + 1$. If $d \leq b$ and $\hat{e}_s = 1$, then append s to S.
- Upon input (spend), set $d \leftarrow d + 1$.
- Upon an input (read), return $\hat{\mathsf{S}}$.

Fig. 6: Description of the desired functionality $\mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,\underline{b}}$.

The "real-world" setting for niPoE protocols. The main difference between the niPoE setting in Section 8 is that we have to describe how the channel $\longrightarrow$ in the three-party scenario is understood. For simplicity, we model it such that if sender P inputs a message into the channel, it can be read by both V and E. Additionally, E can put messages into the channel which will also be readable by V. The shared resource $\mathcal{T}^{\mathrm{RRO}}_{\underline{a},\overline{a},\underline{b}}$ behaves as usual, but allows $\underline{a}$ queries by P, $\underline{b}$

queries by V, and $\overline{a}$ queries by E. That is, the assumed resources are described by $\left[\mathcal{T}^{\mathrm{RRO}}_{\underline{a},\overline{a},\underline{b}}, \longrightarrow\right]$ in this case.

The security definition. Having described the real-world and ideal-world settings, we are now ready to state the security definition. As in the case of a PoE, the definition considers both the case where the prover is honest (this requires that the proof can be performed efficiently) and where the prover is dishonest (this requires that each proof need at least a certain effort).

Definition 12. *A protocol $\pi = (\pi_1, \pi_2)$ is a* non-interactive $(\phi, \underline{b}, \varepsilon)$-proof-of-effort with respect to simulator σ *if for all $\underline{a}, \overline{a} \in \mathbb{N}$,*

$$\pi_1{}^P \pi_2{}^V \left[\mathcal{T}^{\mathrm{RRO}}_{\underline{a},\overline{a},\underline{b}}, \longrightarrow\right] \quad \approx_\varepsilon \quad \sigma^E \mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,\underline{b}} .$$

C.2 Protocol

We prove the statement for the same Construction 3 that we considered in Section 8. We show the following theorem.

Theorem 8. *Let $d \in \mathbb{N}$ the hardness parameter. Then the described protocol (ξ, χ) is a non-interactive $(2^{-d}, \underline{b}, 0)$-proof-of-effort.*

Proof. We have to prove the condition

$$\xi^P \chi^V \left[\mathcal{T}^{\mathrm{RRO}}_{\underline{a},\overline{a},\underline{b}}, \longrightarrow\right] \quad \approx_\varepsilon \quad \sigma^P \mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,\underline{b}} ,$$

with $\varepsilon = \frac{b \cdot 2^d}{|R|}$ and the simulator σ described as follows. Initially, it sets internal query counters to $c_P, c_V \leftarrow 0$. The simulator emulates to E the same interface as $\left[\mathcal{T}^{\mathrm{RRO}}_{\underline{a},\overline{a},\underline{b}}, \longrightarrow\right]$. Whenever E makes a query $\bar{x} \in \{0,1\}^*$ to $\mathcal{T}^{\mathrm{RRO}}$, if $c_P \geq \overline{a}$ then output $\perp$. Otherwise, set $c_P \leftarrow c_P + 1$ and:

- If $\bar{x}$ has been queried before, then respond consistently. (Including cases where the query was made implicitly because the proof was simulated to be sent over $\longrightarrow$.)
- If $\bar{x}$ is not of the form $(s, x) \in \mathcal{S} \times D$, then output a uniformly random element from R.
- If $\bar{x} = (s, x) \in \mathcal{S} \times D$, and for statement s the prover has not yet been solved, activate $\mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,\underline{b}}$ with statement s, and obtain a result bit b. Then:
  - If $b = 1$, sample $\tilde{y}_i \leftarrow\!\!\!\!\!\!\;^{\$}\; \left\{y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^*\right\}$, return $\tilde{y}_i$ and mark s as solved.
  - Otherwise, $b = 0$. Output a uniformly random value from

$$R \setminus \left\{y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^*\right\} .$$

- The final case is that $\bar{x} = (s, x) \in \mathcal{S} \times D$, but the prover has solved for statement s already. In that case, output a uniformly random value from R.

Whenever E sends a message $\bar{x}$ via the channel $\longrightarrow$:

- If $\bar{x} \notin \mathcal{S} \times D \times R$, or $c_V \geq \underline{b}$, then ignore the message. Otherwise, set $c_V \leftarrow c_V + 1$.
- If $\bar{x} = (s, x, y)$ but the response to query (s, x) was not valid (i.e., did not start with 0^d) or not equal to y, or the same input has been given before or it was simulated as a message on $\longrightarrow$, then input (spend) to $\mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,\underline{b}}$.
- If $\bar{x} = (s, x, y)$ and the response to query (s, x) was y, and y starts with 0^d, then invoke copy_s at $\mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,n}$.
- If $\bar{x} = (s, x, y)$ but (s, x) has not yet been queried to $\mathcal{T}^{\text{RRO}}$, then first behave as in a query (s, x) to $\mathcal{T}^{\text{RRO}}$, and subsequently as sending an either correct or incorrect query according to the above rules.

Whenever E attempts to read P's messages from $\longrightarrow$, invoke (read) at $\mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,n}$ to obtain $\hat{\mathsf{S}}$. If a new $s \in \mathcal{S}$ is obtained (or there is still one buffered from a previous invocation), then simulate a valid proof for s as sent over $\longrightarrow$.

In the real-world model, upon each input $s \in \mathcal{S}$ at P, the protocol ξ follows the strategy described in Construction 2. If P has not been successful for $s \in \mathcal{S}$ before, and if the overall number of activations at P is not exceeding $\underline{a}$, the probability of $\mathcal{T}^{\text{RRO}}(s, x') = 0^d|y'$ with $y' \in \{0,1\}^*$ is 2^{-d} since the outputs of $\mathcal{T}^{\text{RRO}}$ are uniformly distributed and independent for differing inputs. Therefore, the probability of returning 1 for an s for which it was not solved before is exactly 2^{-d}, which is exactly the behavior described by $\mathsf{NIPOE}^{\underline{a},\overline{a}}_{\phi,\underline{b}}$. Additionally, in both cases the verifier accepts $s \in \{0,1\}^*$ if the proof is valid and the overall number of verifications does not exceed $\underline{b}$.

Upon an activation at V, if no prover has solved for a (new) statement, then the verifier outputs $\perp$. If a prover has sent a solution $(s', x') \in \mathcal{S} \times D$, then the verifier checks this via $\mathcal{T}^{\text{RRO}}(s', x')$ and outputs s' if the solution was correct.

What remains to be shown is that with the described simulator σ, the two terms in equation Equation C.2 are equivalent. First, the responses to queries to $\mathcal{T}^{\text{RRO}}$ also have the correct distributions.

- Repeated queries are answered consistently in both cases, and queries that are not of the form $(s, x) \in \mathcal{S} \times D$ are answered by a uniformly random value.
- For queries of the form $\bar{x} = (s, x) \in \mathcal{S} \times D$, more care is necessary.
  - If the prover has not yet solved for statement s, the probability of solving is exactly $\phi(j) = 2^{-d}$ by the same argument as in the proof of the two-party setting. Consequently, in the simulation, the probability of returning a (uniformly random) value from the set

    $$S = \left\{ y \in R : y = 0^d | y' \wedge y' \in \{0,1\}^* \right\}$$

    is 2^{-d}, and the probability of returning a (uniformly random) value from $R \setminus S$ is $(1 - \phi(j)) = 1 - 2^{-d}$. As $|R|/|S| = 2^d$, the resulting output is uniformly random in R.
    In the protocol, since $\mathcal{T}^{\text{RRO}}$ has never been queried on the input before, the output is uniformly random from R by definition of $\mathcal{T}^{\text{RRO}}$.

- In case session i has been solved previously, the output is uniformly random from R in both cases.

The read operations on $\longrightarrow$ are also simulated properly. In both cases, in the order in which the proofs were obtained for statements $s_1, s_2, \ldots$ at P, E will observe valid proofs at the channel $\longrightarrow$.

Under the condition that E does *not* send a message (s, x, y) such that $y = \mathcal{T}^{\mathrm{RRO}}(s, x)$, sending a value over the channels $\longrightarrow$ also has the same effects. A value that is not valid, meaning that it is outside of $\mathcal{S} \times D \times R$ or known to not map to a valid solution, does not change the state of the verifier. A value that is known to map to a valid solution will make the verifier accept the corresponding statement. A message (s, x, y) where (s, x) has not been queried to $\mathcal{T}^{\mathrm{RRO}}$ is also treated consistently, namely as first a query to $\mathcal{T}^{\mathrm{RRO}}$ and then as sending the corresponding result to the verifier. This completes the proof. □