Individual Cryptography

Stefan Dziembowski[1], Sebastian Faust[2], and Tomasz Lizurej[1]

[1] University of Warsaw and IDEAS NCBR
[2] TU Darmstadt

Abstract. We initiate a formal study of *individual cryptography*. Informally speaking, an algorithm Alg is *individual* if in every implementation of Alg there always exists an individual user that has full knowledge of the cryptographic secrets S used by Alg. In particular, it should be infeasible to design implementations of this algorithm that would hide the secret S by distributing it between a group of parties using an MPC protocol, or via outsourcing it to a trusted execution environment.
This problem is interesting from a purely theoretic perspective, but it also has applications to preventing attacks called "identity sharing" and "identity lease" that we describe in the paper. We construct a basic primitive within this paradigm called *Proofs of Individual Knowledge*. Our construction is based on the idea that performing quickly a massive amount of hash computations is infeasible without learning significant information about the output of this computation. We provide a formal model for this assumption. Depending on the settings, we believe it is realizable in practice either on standard CPUs (with cryptographic accelerator) or assuming that special-purpose hardware (similar to cryptocurrency mining rigs) is available for the users.

1 Introduction

In this paper, we study the following intriguing question: can one ensure that a given entity on the Internet is a single person, not a "virtual user" emulated by a group of parties using an MPC (multiparty computation) protocol? When parties are identified by cryptographic secrets (e.g., keys or passwords) this problem boils down to checking that a cryptographic secret is known to an *individual* party, and is not, e.g., secret-shared among a set of parties. We formalize this natural question by putting forward a new cryptographic concept that we call *individual cryptography*. Informally speaking, an algorithm Alg is *individual* if in every implementation of Alg there always exists an individual user that has full knowledge of the cryptographic secrets S used by Alg. In particular, it should be infeasible to design implementations of this algorithm that would hide the secret S from any user by distributing it between a group of parties using an MPC protocol, or via outsourcing it to a trusted execution environment (TEE) such as the Intel SGX (see, e.g., [30]).

We instantiate this idea by defining and constructing a new cryptographic primitive called *Proofs of Individual Knowledge (PIK)*. A PIK protocol is run between two parties: a Prover $\mathcal{P}$ and a Verifier $\mathcal{V}$, who both know a message S. The goal of the Prover is to convince the Verifier that she knows S in a very strong sense, namely: S is stored on a single machine $\mathcal{M}$, and it can be recovered from $\mathcal{M}$ at a reasonable cost. Very informally speaking, our solution is based on forcing the Prover to perform intensive computations on the entire S and on measuring her response time. The system's parameters are chosen so that any attempt to distribute the knowledge of S, and to, e.g., emulate the Prover using an MPC protocol, results in a delayed response to the Verifier and hence will be detected by her.

We believe that the question of constructing such a scheme is interesting from a purely theoretical point of view. It comes, however, also with some practical applications. Namely, it can be used as a countermeasure against a new class of attacks called the "identity sharing attacks" that we introduce in this paper. They are inspired by the so-called "identity lease attack" recently identified in [28]. We elaborate on these attacks below. Imagine a service provider $\mathcal{S}$ that maintains a system in which individual users $\mathcal{U}$ can open accounts by paying a subscription fee. In order to lower this fee, the malicious users decide to open a *single* account and to share the credentials S to it between each other. We are interested in situations when these users are individuals $\mathcal{A}_1, \ldots, \mathcal{A}_a$ connected via the Internet that do not trust each other. In other words: we are *not* concerned about scenarios in which the credentials are shared between different devices that belong to

a single person, between members of one family, or between devices that are located physically very close to each other (so the network connection speed does not matter). Suppose that to discourage the users from simply sharing S in plaintext, the service provider integrates ad-hoc countermeasures such that knowledge of S suffices to create significant damage to the account. For example, if the "account" is a cryptocurrency address, then the knowledge of S should suffice to drain the account from all the coins. Alternatively, if it is an online storage system, then knowledge of S should permit deleting all the files.

Since the parties are mutually-distrusting, this countermeasure should suffice to discourage them from sharing S in plaintext. However, it does not offer protection against more sophisticated *identity sharing attacks* as the following example shows. Suppose $\mathcal{A}_1, \ldots, \mathcal{A}_a$ share S using secret sharing and jointly emulate a single "virtual" user $\mathcal{U}$ using an MPC protocol. Note that this attack requires that the entire session between $\mathcal{U}$ and $\mathcal{S}$ is run (on the $\mathcal{U}$'s side) using MPC which may not look very practical. However, with the recent progress in MPC technology it may become a valid threat in the near future.

The identity-sharing attack is similar to the "identity-lease attack" recently introduced in [28]. The main difference is that the attack of [28] relies on the use of so-called *trusted execution environments* (TEEs), such as Intel SGX (see, e.g., [30]). Moreover, in this scenario, it is ok (for the attacker) if one of the users (say $\mathcal{A}_1$) knows S entirely. This attack's main goal is to temporarily outsource ("lease") complete or partial access to the server. Note that if we only state such requirements, we could have a trivial solution where $\mathcal{A}_1$ accesses the server herself and forwards the messages to the other users. This setting is non-trivial because we want the $\mathcal{A}_i$'s to be able to access the server without interacting with $\mathcal{A}_1$. The authors of [28] describe this attack and implement it using the Intel SGX platform. They also discuss some defenses against them, but in general, the message of their work is that such attacks are hard to prevent. As described below, we believe that PIKs can be used to prevent such attacks under the assumption that the parties have access to external hardware that very efficiently computes a large number of hashes in parallel, and that is *not* under the control of a TEE enclave.

1.1 High-level overview of Proofs of Individual Knowledge.

To discuss PIK's security properties, assume that the Prover is emulated by a a *distributed adversary*, i.e., a malicious group of parties $\mathcal{A}_1, \ldots, \mathcal{A}_a$ called the *sub-adversaries*. Intuitively $\mathcal{A}_b$'s do not trust each other, yet they want to emulate a single party. To make the model stronger, we will assume that they only try to cheat each other *passively* (i.e., a passively-secure MPC protocol counts as a valid protocol for the adversaries)[3]. The message S is distributed between the sub-adversaries (in some arbitrary way, e.g, it may be secret-shared, or each $\mathcal{A}_b$ can hold a sub-string of S). Some of the sub-adversaries talk to the Verifier, trying to convince her that S is stored on a single machine.

At first sight, it may be tempting to construct the following trivial protocol for checking the individual knowledge. The Prover simply sends the message S to the Verifier, who exploits some physical properties of the network connection to check if S arrived from one party. It may look like this solution works in practice since violating this check would require contrived attacks that split a physical communication session into parts, with, e.g., one sub-adversary sending the first half of S to the Verifier and the second one sending the Verifier the second half of it.

At a closer look, however, this solution is not what we are looking for. The reason is that in typical practical applications, we need S to be secret; hence, S cannot be sent in plaintext over the network. Therefore, in the solution described above, S would need to be encrypted (or hidden in some other cryptographic way). If it is encrypted by some long-term key shared between $\mathcal{P}$ and $\mathcal{V}$, any sub-adversary $\mathcal{A}_b$ could simply store the ciphertext C of S (instead of storing S), and send C to $\mathcal{V}$. One can try to fix this problem by adding randomness in the communication, i.e., letting $\mathcal{V}$ send to $\mathcal{P}$ a random nonce Z, and making C dependent on Z. This also does not work since the sub-adversaries can secret-share S between themselves (using some a-out-of-a secret sharing scheme), and then use an MPC protocol to compute C from S and Z. If the number

[3] Clearly, going to the active case would make the model weaker. Moreover, we would need to deal with problems coming from the fact that the $\mathcal{A}_b$'s could use techniques for punishing each other for cheating (e.g. [2, 8, 29, 4])

of sub-adversaries is small (e.g. $a = 2$), then the current techniques, can be used to compute C very efficiently (for popular encryption schemes), see, e.g., [26].

In the example above, we used the fact that long as an MPC protocol emulating $\mathcal{P}$ can be executed in time comparable to the network latency, the Verifier cannot distinguish between talking to a single machine or a distributed adversary. This observation can be generalized: it is easy to see that no PIK can have a very efficient Prover, where by "very efficient" we mean: one that can be implemented using an MPC protocol in time comparable to the network latency. Consequently, we need the Prover to perform some moderate amount of computational effort. This work has to be small enough to be quickly doable in a non-distributed way and large enough to guarantee that computing it using an MPC protocol takes a substantial amount of time. Jumping ahead, we will achieve it by a technique similar to the Bitcoin puzzles (see, e.g., [20]).

Regarding the confidentiality of S we also need to clarify the following. Our basic definition of PIK (see Def. 1) does not guarantee any confidentiality for S, and in particular, it may look like the trivial protocol described above (just sending S may actually satisfy it) could satisfy it. It is important to stress that this is not true. In short: this is because, in our security definition, we do *not* assume that the messages sent (or received) by a sub-adversary $\mathcal{A}_b$ are known to $\mathcal{A}_b$. As we explain in Sect. 1.2, the only information that the sub-adversaries "know" are the outputs of the hash function that they compute. This allows us to use the PIK protocol that we construct in Sect. 2.3 in more practical scenarios described in Sect. 3.1, in which the communication is encrypted, or when the privacy of S is protected using the zero-knowledge techniques. Finally, let us stress that in our definitions, we do not assume anything about the distribution of S, although in reality, in many applications, S would be a cryptographic key, and hence it would have high entropy or even be uniform.

1.2 Informal description of our model

Let us start with a discussion on how to model the fact that a machine "knows" some secret S. The question of formalizing knowledge has a long history in cryptography, and in particular, it has been studied in the context of "proofs of knowledge" [5], "knowledge of exponent" [14], or "plaintext-awareness" [6]. None of these approaches considers attacks by a distributed adversary. The most relevant to ours is the model of [6]. The authors of this paper assume existence of a hash function modeled as a *random oracle* Ω_H [7], which is an external entity storing a randomly-chosen function $H : \{0,1\}^* \to \{0,1\}^\kappa$ (where 1^κ is a security parameter). It is assumed that if an adversary $\mathcal{A}$ evaluated H on some input x then $\mathcal{A}$ knows the input x and the corresponding output $H(x)$. Technically: the (input, output) pairs are later given to an algorithm $\mathcal{E}$ called "knowledge extractor". If based on these tuples, the knowledge extractor outputs some message S, then we assume that "$\mathcal{A}$ knows S" (since $\mathcal{A}$ could have computed S herself just by observing the oracle queries and the replies to them).

Initial attempts. Let us try to adapt the model of [6] to our setting. We will now present some examples that illustrate problems with such adaptation (but eventually lead us to our solution). First, assume that every query to the random oracle comes from some sub-adversary $\mathcal{A}_b$, and there is a separate knowledge extractor $\mathcal{E}_b$ for each $\mathcal{A}_b$. The extractor $\mathcal{E}_b$ gets as input the oracle queries of $\mathcal{A}_b$ and tries to extract S from them. One obvious idea is to simply say that a PIK is secure if for every distributed adversary $\mathcal{A}_1, \ldots, \mathcal{A}_a$ (that successfully emulates $\mathcal{P}$) there exits an extractor $\mathcal{E}_b$ that produces S as output.

Unfortunately, this approach needs some important modifications before we can use it for modeling knowledge in our setting. To illustrate the first modification consider the following candidate protocol for a PIK: the Verifer sends a random nonce Z to the Prover, and the Prover answers with $H(S||Z)$ (the Verifier accepts if the result is correct). Since computing popular hash functions in MPC can be done very efficiently, the Prover's answer can be computed quickly even if S is distributed (between 2 parties, say). Hence, this solution is *not* secure in practice, even though it can be proven secure in the model described above. Indeed, since in this model $(S||Z)$ has to be submitted to the random oracle by *some* sub-adversary $\mathcal{A}_b$, it is given to the knowledge extractor $\mathcal{E}_b$ as input. This extractor can now trivially output S by stripping Z off.

The moral of this example is that we need to model the fact that a certain number of queries can be computed using an MPC protocol. We call such queries *slow* and assume that the adversary has a limited

budget for them. We denote this budget of slow queries with σ. Informally, it corresponds to the "number of MPC computations of H that the sub-adversaries can perform without significantly delaying their responses in the protocol". The slow queries (indicated with a special flag *mode* being set to slow) will be "invisible" from the point of view of the knowledge extractors. The remaining queries will be called *fast* (and will have *mode* := fast). We will assume that the sub-adversaries can perform a large number of them (only bounded by their polynomial running time).

Unfortunately, this still does not suffice for modeling the security of PIKs in a realistic way. For illustrative purposes, let us consider the following idea for a PIK, inspired by how the Bitcoin puzzle work. Let S be the message. The Verifier picks up a random challenge $Z \in \{0,1\}^\kappa$ and sends it to the Prover, who has to find W such that $H(S \,||\, Z \,||\, W)$ starts with ζ zeros when written in binary (where "$||$" denotes the concentration of string). This is done by trying multiple values for W until the right one is found. Here, ζ is chosen in such a way that computing 2^ζ times the hash function (i.e. the expected number of hashes needed to find one that starts with ζ zeros) takes some moderate amount of time on a single machine and much more time using an MPC protocol. Or in other words: ζ needs to be such that the adversaries do not have the budget of slow queries to compute 2^ζ hashes (i.e., $\sigma \ll 2^\zeta$). The fact that S is stored on a single machine could then be checked by measuring the response time of the Prover.

The above approach, however, does not work for many natural choices of H. This is because a typical hash function has a structure that can be exploited to distribute its computation. In particular, popular hash functions operate by dividing the input message S into blocks $(S_1, \ldots, S_s)$ and then processing it block-by-block via a so-called mode of operation. This holds both for the hash functions based on the Merkle-Damgard paradigm and for the ones using the sponge construction (see, e.g., [23] for more on this). For concreteness, suppose H is a Merkle-Damgard-type hash function, and let $G : \{0,1\}^{2\kappa} \to \{0,1\}^\kappa$ be the compression function that H is built from. Now suppose that we want to verify if a message S consisting of two blocks (S_1, S_2) (both of length κ) is stored on one machine. Taking into account the structure of H, the Prover has to compute an expected number 2^ζ of the following hashes (for fixed (S_1, S_2, Z) and different W's): $h = H(S \,||\, Z \,||\, W) = G(G(G(G(IV, S_1), S_2), Z), W)$ (where $IV \in \{0,1\}^\kappa$ is some fixed initial value, and for simplicity, we omit the last block encoding message length). Now imagine two sub-adversaries: $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively holding S_1 and S_2. They can easily compute H without communicating their S_i's to each other, by $\mathcal{A}_1$ simply sending a value $g := G(IV, S_1)$ to $\mathcal{A}_2$, and then $\mathcal{A}_2$ computing $h := G(G(G(g, S_2), Z), W)$. Moreover, once g is sent to $\mathcal{A}_2$ she can use it to quickly solve the puzzle by computing $G(G(G(g, S_2), Z), W)$ for different values of W and without interacting with $\mathcal{A}_1$. Hence, this PIK protocol is in practice *not* secure. On the other hand (assuming that $\sigma \ll 2^\zeta$), this protocol can be proven secure in a simplified model which does not consider the structure of the hash function.

We consider two approaches for dealing with this problem. The first one is to look at H as a mode of operation for the compression function G, treating G as an "indivisible" building block in our constructions. Such approach is often used, e.g., in the literature on space-bounded cryptography model. In our context, it would work as follows. Suppose a sub-adversary $\mathcal{A}_b$ evaluated G on input x. In this case, the corresponding extractor $\mathcal{E}_b$ learns x entirely. Note that this approach is rather optimistic, as it assumes that G does not have any internal structure that could be exploited, by, e.g., $\mathcal{A}_1$ precomputing G on a part of x and sending the output of this precomputation to $\mathcal{A}_2$. Nevertheless it can work in practice for some real-life hash functions.

In this case, the construction of PIK is rather straightforward. More concretely assume that that H is a Merkle-Damgard-type hash function based on compression function $G : \{0,1\}^{2\kappa} \to \{0,1\}^\kappa$. Then, one can take the "PoW' idea described above and modify it by changing the order of the values that are hashed: instead of evaluating H on $(S \,||\, Z \,||\, W)$, evaluate it on $(Z \,||\, W \,||\, S)$, and assume that $Z, W \in \{0,1\}^\kappa$. In this way, S cannot be divided in separate substrings on which H is precomputed, unless the sub-adversaries communicate large amounts of data. For example, if, as above, $S = S_1 \,||\, S_2$ (with $S_1, S_2 \in \{0,1\}^\kappa$) and each S_i is held by $\mathcal{A}_i$ (for $i \in \{1,2\}$) then computing $H(Z \,||\, W \,||\, S)$ for each W would require $\mathcal{A}_1$ to send $H(Z \,||\, W \,||\, S_1)$ to $\mathcal{A}_2$. Hence, the number of messages sent between the sub-adversaries would be linear in the number of hashes that are computed.

In many scenarios, quickly sending such large amount of data is infeasible, and hence the response time of the distributed adversary would be much longer than the response time of the honest Prover. One can make

this gap even larger, by increasing the round complexity of the computation of H by a distributed adversary. Let ρ be a natural number denoting the maximal number of rounds of communications between the sub-adversaries. Then, computing $H\left(Z \,||\, (W \,||\, S)^{\rho+1}\right)$ is infeasible without letting one of the sub-adversaries know S entirely.

The details of this model and the proof of the security of this construction will be provided in the extended version of this paper.

Our model. As remarked above, the assumption that "if G has been quickly computed then someone knows the entire input of G" is rather strong. In this paper paper, we propose an alternative, much weaker assumption that totally abstracts away from the structure of the hash function. More precisely, we assume that only the *outputs of H are known to the sub-adversary who evaluated H* (using a fast query). In other words, we assume that if a distributed adversary computed a large amount of hashes H then for a substantial number of them their output needs to be such that at least one sub-adversary $\mathcal{A}$ knows it, which is modeled by giving this output to the knowledge extractor $\mathcal{E}_b$. We believe that the assumption of only knowing the output is the most conservative assumption that can be made, and in particular, it includes models where the adversary knows the inputs to the compression function. Our formal model appears in Sect. 2.1. Below, we discuss some natural questions related to our security model.

How can a "normal user" recover S. It is natural to ask how an individual user (lacking technical background) can recover S when a knowledge extractor (that may be a sophisticated algorithm) can potentially recover S from the hash outputs. We stress, however, that our definition makes this process fully automatic since S is computed by the extractors that are algorithms. In practice, one can imagine that the entity that is interested in S not being distributed (e.g., the service provider) creates and publishes on the Internet computer programs that work as extractors or even pretends to be one of the users to find people willing to jointly cheat the service by creating a distributed identity. Another option is to create a third-party service that helps in extracting S from the execution transcript of $\mathcal{A}_b$. Note also that in many applications (e.g., credential sharing) the sheer fact that a malicious user *risks* that S leaks would discourage them from cheating.

Computing hashes in hardware. Several modern CPUs are equipped with cryptographic hardware accelerators that compute hashes in hardware much faster than in software. This potentially gives even stronger guarantees for the PIK protocols constructed within our model since it allows us to set the parameter denoting the number of hashes computed by an honest Prover (in the examples above, this was 2^ζ) to a higher value. Also, it may make it harder to break our schemes using the "heuristic MPC" attacks (see below) since the calls to the hash function are better isolated.

Security against TEE. A possible way to attack PIK is to use trusted execution environments (such as Intel's SGX) to accelerate the MPC (see, e.g., [3]), or (in case of the identity-lease attack), to outsource S in a way similar to the one described in [28]. In general, it looks hard to prevent such attacks in the presence of strong hardware assumptions since a third party that is fully trusted by all the users $\mathcal{A}_b$ trivializes their task of securely sharing the credentials S and authenticating using them. Our solution to this problem is to introduce an additional element into the system. Namely, we assume that the honest users are equipped with hardware similar to Bitcoin mining rigs that can compute a massive amount of hashes in parallel.[4] Our assumption is that computing large amounts of hashes very quickly within a TEE enclave is infeasible. This assumption seems true for the currently used TEEs since they have only as much power as the CPU they run on and are limited in memory.

[4] While it is not clear if the current mining rigs could be adapted to this setting due to being highly optimized for solving Bitcoin's PoW, at least their existence provides some estimates on what numbers of "hashes per second" are achievable at what price. At the moment of writing this text, a mining rig called "AntMiner S9" computing $1.35 \cdot 10^{13}$ hashes per second costs around USD 750.

Attacks by "heuristic MPC". One obvious idea to attack the assumptions we make in our model is to use some "heuristically-secure MPC" techniques that are more efficient than a fully-secure MPC protocol typically considered in the cryptographic literature. By "heuristically secure," we mean protocols that do not come with formal security guarantees but make it hard in practice to recover the inputs (e.g., by using heuristically secure code obfuscation). We address this concern in the following way. First, to attack our assumption such a heuristic MPC would need to be comparably efficient to computing hash functions "in plain". Secondly, our assumption that only the output of a hash function H is known to the adversary means that it is enough for her to know any intermediary values used in the computation of H that are sufficient to compute the hash output. Hence, obfuscation would need to be applied to the entire computation of H. Moreover, in some applications, one can imagine external services that help parties to de-obfuscate the computation (this makes sense, especially if breaking the scheme can result in earning money that was put aside as a deposit). Finally, let us remark that relying on highly optimized hardware computation of hashes makes such heuristic obfuscation harder.

1.3 Informal description of our solution

As outlined above, a PIK protocol should allow the Prover $\mathcal{P}$ to convince the Verifier $\mathcal{V}$ that a secret S (that they both know) is stored on one machine, and it is known to the machine owner. This is done by forcing $\mathcal{P}$ to quickly reply to challenges Z from $\mathcal{V}$ in a way that proves that $\mathcal{P}$ performed intensive computation on the entire value of S. Since $\mathcal{P}$ measures the response time, it will notice any response delays that are due to the fact that S is, in fact, distributed between different "sub-adversaries", and it is not stored on one machine. This is done by performing "hash computations" on S. We consider both the case when $\mathcal{P}$ has a regular CPU (when PIK is a countermeasure that is sufficiently strong against the attacks by MPCs) and when $\mathcal{P}$ is equipped with a "mining rig" that allows $\mathcal{P}$ to compute a large number of hash computations (which is a countermeasure also against the TEEs). Let us also stress that in our definition we will tolerate that the knowledge extractors output more than one candidate message S, i.e., we permit many "false positives". We outline a method for eliminating them in Sect. 3.2.

The two main parameters that characterize the adversary are σ – the number of hashes that can be computed in such a way that no sub-adversary learns their outputs since they are computed using MPC or TEE (in our model, this corresponds to the "slow" queries to the oracle), and ρ – the number of communication rounds between the $\mathcal{A}_b$'s. In Sect. 1.2, we already explained the idea behind the slow queries. The bound on the number of communication rounds is quite mild in practice. Recall that $\mathcal{A}_b$'s are connected over the Internet, and hence assuming that no more than 1000 rounds of communications per second (say) are executed between them is reasonable.

Our scheme is described formally in Sect. 2. The reader may, in particular, look at the diagram in Fig. 2 – we will be referring to it while presenting our solution informally below. Recall that in our model, we assume that only the outputs of the (fast) hash computations are known to some sub-adversary $\mathcal{A}_b$. Let us start with describing a simple scheme for proving knowledge of 1-bit messages (i.e., messages of a form $S = (S_1)$), and assuming that (a) the sub-adversaries cannot execute any slow queries, and (b) the sub-adversaries cannot communicate during protocol execution. In this case, the following idea works: the Verifier sends a challenge Z to the Prover, and the Prover has to respond with $h = H(H(Z) \,||\, S_1)$. Since no slow queries are allowed, and the sub-adversaries cannot communicate, thus one sub-adversary, $\mathcal{A}_b$, say, needs to know both outputs $h' := H(Z)$ and h, and can now extract S_1 by checking if $h = H(h' \,||\, 0)$ or $h = H(h' \,||\, 1)$. This, of course, works under a very artificial assumption that $\mathcal{A}_b$ does not compute other hashes and that the knowledge extractor knows which hash value corresponds to h and which to h'. In our model, we do not make such assumptions, but for these informal explanations, let us use them.

At first sight, the above protocol may look somewhat artificial. In particular, the reader may feel like the need to "hash twice" (first to compute h' and then h) is just a consequence of the model definition and has no practical meaning. We argue that this is not the case. The reason for forcing the Prover to compute $H(H(Z)\,||\,S_1)$ instead of simply computing $H(Z\,||\,S_1)$ is that in the former case, we make the Prover perform non-trivial computations on Z, *before* using the output of this computation (h') to compute the final response (that also depends on S_1). The point is that while the sub-adversaries may manage to communicate with

$\mathcal{V}$ in such a way that none of them learns Z, it is much less likely that the sub-adversaries will be able to quickly compute $H(Z)$ without one of them learning $H(Z)$. Hence, S_1 should be much easier to recover by practical knowledge extractors.

Let us now show how to remove our artificial assumptions in the above example.

Allowing slow queries. Recall that above we assumed that the sub-adversaries could not perform any slow queries (i.e., no H can be computed using MPCs or a TEE). We eliminate this restriction in the following way (already discussed above): namely, we force the Prover to perform multiple computations of hashes on different nonces to find a nonce that leads to a hash starting with ζ zeros. This is very similar to Bitcoin's puzzles, except that we do it κ times to reduce the variance for finding a solution. The nonce that used in the ith puzzle is denoted with W^i.

Longer messages S. Let us now discuss how to eliminate the assumption that S has just one bit. Let $S = (S_1, \ldots, S_n)$ where $n \geq 1$. Our idea is straightforward (see also the first column on Fig. 2, ignoring the values at the beginning of the hashed blocks, i.e., $0, 1, \ldots, n$ and i): we apply the construction for a single bit iteratively, i.e., for a the ith nonce W we let $Q_1 := H(Z \,||\, W)$, and then for each $j = 2, \ldots, n+1$ we let $Q_j := H(S_j \,||\, Q_{j-1})$.

Allowing communication between the sub-adversaries. Note that the above construction is insecure if there are no restrictions on the communication between the $\mathcal{A}_b$'s. Indeed, imagine two sub-adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ and suppose S has two bits $S = (S_1, S_2)$, with $\mathcal{A}_1$ holding S_1 and $\mathcal{A}_2$ holding S_2. Then these adversaries can break the above PIK as follows: $\mathcal{A}_1$ computes a massive number of hashes $Q_2 := H(S_1 \,||\, H(Z, W))$ (for different values of W), and sends the Q_2's to $\mathcal{A}_2$. Sub-adversary $\mathcal{A}_2$ processes every Q_2 by computing $Q_3 := H(S_2 \,||\, Q_2)$ in order to find Q_2 that is such that Q_3 starts with ζ zeros. Once such Q_2 is found, she communicates it to $\mathcal{A}_2$, who checks which W this Q_2 corresponds to, and sends this Q to the Verifier.

The above example shows that we need to restrict communication between the sub-adversaries. We choose to do it by putting a bound ρ on the number of rounds (an alternative approach would be to bound the communication size, but it seems more challenging to work with in practice). In our construction, we use this assumption by requiring that the Prover needs to compute $d = 2\rho$ iterations of the procedure outlined above (cf. 2).

Outlook. Our final construction is presented in detail in Sect. 2.3 and on the diagram on Fig. 2. While this construction may not look very natural, we would like to stress that it can be implemented quite simply as an iterative application of a Merkle-Damgard hash function to an encoded version of S. We explain it in more detail in Sect. 2.3.

1.4 Uses cases

As already mentioned, our motivation is primarily theoretical, yet we believe some use cases for our ideas exist. Let us briefly describe them. The first one is a mechanism for identity-sharing prevention. In this case, the service provider checks if the credentials S are stored on one machine by regularly running the PIK protocol (typically with a mechanism for ensuring the privacy of S described in Sect. 3.1). If we are not concerned by the attacks of TEE, then we can assume that the honest Prover runs on a regular CPU, as most likely, we can choose parameters ρ and σ in such a way that distributing the Prover is infeasible. If we want to address the TEE attacks, then probably the best way to guarantee that the Prover emulation cannot be distributed is to make Prover's computation very intensive by assuming that she is using external machines (similar to cryptocurrency "mining rigs") that compute hashes faster than the CPU that has the TEE system (see *Security against TEE* on page 5)

The second use case is protection against the identity lease [28]. Recall that in this attack, $\mathcal{A}_1$ knows S and outsources it to $\mathcal{A}_2, \ldots, \mathcal{A}_n$ so that they can non-interactively access the service without learning S. This attack is based on the TEEs and works in non-distributed settings, i.e., when outsourcing is just for one party $\mathcal{A}_2$. The whole point of the attack is to prevent $\mathcal{A}_2$ from learning S. PIK can be used to guarantee

that a party using the service knows S (hence the above attack does not work). This probably requires the use of the "mining rigs" mentioned above.

We leave the analysis of the practicality of the TEE attacks as future work. Note also that multiple attacks against TEEs have been described (see, e.g., [19]). It would be interesting to examine how practical these attacks are in the case of TEEs being used to hide information about very intensive hash computation.

1.5 Other related work

Using cryptography for malicious purposes has been studied before, most notably in the context of "Cryptovirology" proposed by Young and Yung [31]. The approach of [31] focuses on the malicious use of public-key encryption, and not the MPCs. Hence, our paper can be viewed as a natural continuation of the approach of [31] (with MPCs being more "advanced" primitives than the public-key encryption schemes). The idea of preventing leaking secrets has been studied extensively in a context such as traitor-tracing, e.g., it [13, 24]. Up to our knowledge, none of these works considers a distributed adversary.

On a higher level, our paper is also related to papers that look at the question of defining the notion of "identity" in cryptography. In particular, it has some similarities to Position-Based Cryptography [12], where an identity of a user is defined by its geographic location. This approach is also based on measuring Prover's response time, but it is assumed that the communication is not done over the Internet but via physical signals (the whole approach is based on the fact that the speed of electromagnetic signals is fixed). Another difference is that the users in [12] do not have secret credentials but are identified by their geographic location. As already mentioned, our proof techniques are similar to those used in the context of space-bounded cryptography, in particular in the construction of schemes that are secure based on assumptions about the restricted memory of the adversary and whose security relies on hash functions, see, e.g., [16, 17, 18, 1]. For the differences between our model and the ones used in this area, see Sect. 1.2.

1.6 Preliminaries

An empty string is denoted with $\perp$. For a real number x let $\exp(x) = e^x$, where e is the Euler's number. We will use the following versions of the Chernoff–Hoeffding bounds (see, e.g., Thm. 1.1 in [15]).

Lemma 1. *Let $U_1, \ldots, U_C$ be random variables independently distributed over $[0,1]$ and let $U = U_1 + \cdots + U_C$. Then: (1) for all $\delta > 0$ we have $\Pr[U > \mathbb{E}[U] + \delta] \leq \exp(-2\delta^2/n)$ and (2) for all $\epsilon > 0$ we have $\Pr[U < (1-\epsilon)\,\mathbb{E}[U]] \leq \exp(-\epsilon^2 \cdot \mathbb{E}[U]/2)$.*

2 Proofs of Individual Knowledge

We now provide formal details of the definition and the construction that were informally presented in Sect. 1. For the reference, the main notation used in this section is summarized on Fig. 4 in the Appendix (see page 22).

2.1 The model

This section provides more details on the model already informally introduced in Sect. 1.2. All protocols are executed in an asynchronous model. Every protocol is parameterized by a security parameter 1^κ and a hash function H that is modeled differently in the honest and in the adversarial executions. In the honest execution the parties access H in a black-box way (via a standard random oracle [7]), except of Sect. 3.1, where a "circuit access" to H is needed (in the construction of zkPIK).

In the adversarial model, the malicious parties access H via an interactive machine Ω_H^{MPC}, called an *MPC oracle*, which chooses a random function $H : \{0,1\}^* \to \{0,1\}^\kappa$ and interacts with parties $\mathcal{A}_1, \ldots, \mathcal{A}_a$ by accepting queries of a form $(x, mode)$, where $x \in \{0,1\}^*$. Each such a query, coming from a party $\mathcal{A}_b$ is answered to $\mathcal{A}_b$ with $H(x)$ (we also say that $\mathcal{A}_b$ *evaluated* H on input x). We say that Ω_H^{MPC} *is σ-bounded*

if the total number queries answered by the oracle with $mode = \mathsf{slow}$ is at most σ. The queries that exceed this quota are answered with $\perp$. The total number of queries with $mode = \mathsf{fast}$ is only bounded by the time complexity of the adversaries (i.e., it is polynomial in κ).

The main idea is that queries with $mode = \mathsf{fast}$ are "cheap" and the participants will be allowed to send much more of them than the "expensive" queries with $mode = \mathsf{slow}$ (which correspond to queries computed using MPC/TEE techniques). On the other hand, when analyzing what the adversarial parties learned from the execution (i.e. when defining the "knowledge extractors", see below), only the queries with $mode = \mathsf{fast}$ will count. Namely, only the replies to such queries will be considered known to the querying party. Note that the *mode* flag is only used for "accounting" purposes: the actions Ω_H^{MPC} do not depend on the value of this flag, except when defining the available budget of queries.

At the end of the execution of a protocol, we look at the information each party received as a result of the fast oracle queries. We define the *local hash-output transcript of a party* $\mathcal{A}_b$ to be the sequence OUT_b of messages T it received as a result of the fast queries (in the same order in which they were received). A *(knowledge) extractor* $\mathcal{E}_b$ *for* $\mathcal{A}_b$ is a deterministic poly-time machine that takes OUT_b and produces as output a finite set $\mathcal{E}_b(OUT_b) \subset \{0,1\}^*$ (we also say that $\mathcal{E}_b$ *extracted* $\mathcal{E}_b(OUT_b)$). The extractors have a block-box access H (via the same oracle Ω_H^{MPC}, only using the fast queries). We will say that $\mathcal{E}_b$ has *hash complexity* c if it makes at most c queries to H.

2.2 Definition

A *Proof of Individual Knowledge (PIK)* is a protocol π_{PIK} between a Prover $\mathcal{P}$ and a Verifier $\mathcal{V}$ (also denoted $(\mathcal{P} \leftrightarrows \mathcal{V})$). The Prover and the Verifier take as input a pair $(1^\kappa, S)$, where $S \in \{0,1\}^n$ (for some parameter n). Let $h_{\mathcal{P}}$ and $h_{\mathcal{V}}$ be the maximal number of times the Prover and the Verifier (respectively) evaluate H. We require that the protocol is *complete*, i.e., if both parties are honest (and their inputs are as above), then with overwhelming probability, the Verifier outputs yes (in which case we also say that $\mathcal{V}$ *accepts*). The second required property is *soundness*. Define a (ρ,σ)*-distributed adversary* to be a tuple $(\mathcal{A}_1,\ldots,\mathcal{A}_a)$ of poly-time interactive *sub-adversaries*. The honest Verifier $\mathcal{V}$ receives $(1^\kappa, S)$ as input and interacts with $\mathcal{A}_1$ that also receives $(1^\kappa, S)$ as input. Intuitively, $\mathcal{A}_1$, plays the role of the (malicious) Prover from the point of view of the Verifier $\mathcal{V}$. The $\mathcal{A}_b$'s interact with the σ-bounded oracle Ω_H^{MPC}. This interaction is divided into at most ρ rounds, each of them being of the following form: (1) each sub-adversary $\mathcal{A}_b$ performs some local computation, at the end of which $\mathcal{A}_b$ outputs string Str_b and (2) each Str_b is delivered to every other sub-adversary $\mathcal{A}_{\widehat{b}}$. The model above ignores the issue of the length of communicated messages. This is because our results rely only on the bound on the number of rounds. Note that we also assume (in Step (2)) that the messages are communicated by a "broadcast channel" (each Str_b can be seen by all the sub-adversaries). This is ok, since we do not need to be concerned by the privacy of the messages, as the messages received by each $\mathcal{A}_{\widehat{b}}$ do not count as the "knowledge of $\mathcal{A}_{\widehat{b}}$" (more precisely: it is not be given to the extractors as input). Recall also that we assume that the sub-adversaries are performing *passively*-secure computation, and hence guaranteeing broadcast consistency is not an issue.

Consider an execution of a distributed adversary $(\mathcal{A}_1,\ldots,\mathcal{A}_a)$ against an honest Verifier (on input $(1^\kappa, S)$). Define $\mathsf{exec}((\mathcal{A}_1,\ldots,\mathcal{A}_a) \leftrightarrows \mathcal{V}); 1^\kappa; S)$ to be equal to $(OUT_1,\ldots,OUT_a,OUT_{\mathcal{V}})$, where each OUT_b is the local hash-output transcript of $\mathcal{A}_b$ and $OUT_{\mathcal{V}} \in \{\mathsf{yes},\mathsf{no}\}$ is the output of $\mathcal{V}$. We now have the following.

Definition 1. *We say that* π_{PIK} *is a* PIK protocol (for messages of length n) secure against (ρ,σ)-distributed adversary *if there exists knowledge extractors* $\mathcal{E}_1,\ldots,\mathcal{E}_a$ *such that for every* (ρ,σ)*-distributed adversary* $(\mathcal{A}_1,\ldots,\mathcal{A}_a)$ *and every* $S \in \{0,1\}^n$ *we have that*

$$\Pr[OUT_{\mathcal{V}} = \mathsf{yes} \text{ and } S \notin \mathcal{E}_1(OUT_1) \cup \cdots \cup \mathcal{E}_a(OUT_a)\}] \leq \mathsf{negl}(\kappa), \tag{1}$$

where $(OUT_1,\ldots,OUT_a,OUT_{\mathcal{V}}) \leftarrow\!\!\$\ \mathsf{exec}((\mathcal{A}_1,\ldots,\mathcal{A}_a) \leftrightarrows \mathcal{V}); 1^\kappa; S)$. *We say that* π_{PIK} *has* extraction efficiency $(\alpha_{\mathsf{O}},\alpha_{\mathsf{T}},\alpha_{\mathsf{S}})$ *if* $|\mathcal{E}_b(OUT)| \leq \alpha_{\mathsf{O}}$*, and every* $\mathcal{E}_b$ *operates in time at most* α_{T} *and uses space at most* α_{S}*. The parameters* $\alpha_{\mathsf{O}},\alpha_{\mathsf{T}}$ *and* α_{S} *can be functions of some other parameters in the system.*

2.3 Construction

In this section, we present our construction of a PIK protocol π_{PIK}, which was already informally described in Sect. 1.3. The protocol is parameterized by a "moderate hardness parameter" $\zeta \in \mathbb{N}$ and a security parameter 1^κ. We assume the binary representations of natural numbers are of length $\max(\lceil \log_2 n \rceil, \lceil \log_2 \kappa \rceil)$ (where n is the length of the input message S), and write $\mathsf{bin}(i)$ to denote such a representation of an integer i. Our PIK protocol is depicted on Fig. 1 (b). The protocol starts with the Verifier sending a random challenge Z. Then, the goal of the Prover is to find κ nonces $W^1, \ldots, W^\kappa$ that convince the Verifier that the Prover did substantial amount of work for the challenge Z. As explained in Sect. 1.3, this is similar to Bitcoin mining procedure, except that we require κ nonces to be found (in Bitcoin $\kappa = 1$) in order to reduce the variance. The search for the W^i's can be fully parallelized. The only non-parallelizable part is the $\mathsf{scratch}$ procedure described below (see also Fig. 2 for a graphical representation of the computation of $\mathsf{scratch}$).

The $\mathsf{scratch}$ procedure takes as input (i, S, Z, W), where i is the puzzle index, S is the message, Z is the challenge and W is the nonce. The main idea behind the $\mathsf{scratch}$ procedure is that it forces the computing party to sequentially compute d times H on each "encoded" bit of S (see also Sect 1.3 for a an informal description of this procedure). Call a computation of each $Q_1^k, \cdots, Q_n^k$ within a $\mathsf{scratch}$ procedure a *mini-round* (note that each mini-round is represented as a single column on Fig. 2). We call a given scratch *successful* if it starts with ζ zeros (cf. Step 2b on Fig. 1 (b)). Note that in total $\mathsf{scratch}$ computes $nd + 1$ hashes. This is because since $|S| = n$, on each of the bits of S we perform d hash evaluations (one per mini-round), plus we also have one initial hash computation on $(\mathsf{bin}(0) \,||\, i \,||\, Z \,||\, W)$).

Implementation issues. As this is mostly a theoretical paper, we do not focus too much on the implementation issues. Let us only observe that the $\mathsf{scratch}$ function can be represented as follows. Suppose $\widehat{H}$ is a hash Merkle-Damgard function and $H : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}^\kappa$ is the compression function that it is built from. Consider an encoding that takes a message $S = (S_1, \ldots, S_n)$ and encodes it to n blocks (each of length κ) as $\mathsf{Enc}(S) := (B_1 \,||\, \cdots \,||\, B_n)$, where each $B_j = (0^c \,||\, \mathsf{bin}(j) \,||\, S_j)$ (and c is chosen in such a way that the length of B_j is κ). Then, Steps 2b and 2b of the $\mathsf{scratch}$ procedure (Fig. 1 (a)) can be rewritten as $Q_n^k := \widehat{H}(\mathsf{Enc})$ (above we ignore the issue of the IV and length encoding block in the Merkle-Damgard transform). Hence, essentially the $\mathsf{scratch}$ procedure can be represented as interatitive application od $\mathcal{H}$ to it's own output and to $\mathsf{Enc}(S)$. Security analysis of our protocol is provided in the following lemma.

Lemma 2. *Fix an arbitrary message length $n = |S|$, the maximal number σ of "slow queries", and the maximal number ρ of rounds. For a security parameter 1^κ let $d := 2\rho$ and $\zeta := \lceil \log_2(\sigma/(\rho \cdot \kappa)) \rceil + 5$. Then for $\eta_{\mathcal{P}} = (nd+1) \cdot 2^{\zeta+1} \cdot \kappa$ and $\eta_{\mathcal{V}} = (nd+1) \cdot \kappa$, the protocol π_{PIK} from Fig. 1 is a PIK protocol against a (ρ, σ)-distributed adversary with extraction efficiency $(\alpha_{\mathsf{O}}, \alpha_{\mathsf{T}}, \alpha_{\mathsf{S}})$, where*

$$\alpha_{\mathsf{O}} = 2^{14-2\zeta} \cdot \ell^2/(d^2 \cdot \kappa),\ \alpha_{\mathsf{T}} = O(n \cdot \Phi_b),\ \text{and}\ \alpha_{\mathsf{S}} = O(2^{-\zeta} \cdot \ell/d + \Pi_b).$$

Above, ℓ is the number of hashes computed by all the sub-adversaries, and for each b the value Φ_b is the running time of $\mathcal{A}_b$ and Π_b is the space complexity of $\mathcal{A}_b$.

The proof of this lemma is presented in Sect. 2.4. Before proceeding, let us comment on the parameters in the lemma statement. When it comes the parameters of the honest Prover and Verifier, the most important ones are those denoting the "budget" for the hash computations, i.e., $\eta_{\mathcal{P}} = (nd+1) \cdot 2^{\zeta+1} \cdot \kappa$ and $\eta_{\mathcal{V}} = (nd+1) \cdot \kappa$ respectively. Note that each computation of the $\mathsf{scratch}$ procedure requires $(nd+1)$ hash computations. The Verifier needs to do such a computation κ times; hence, she needs to perform only $(nd+1) \cdot \kappa$ hashes. For the Prover, observe that each scratch attempt succeeds with probability $2^{-\zeta}$ (where by "succeeding" we mean finding a value that starts with ζ zeros). Since the Prover needs to be successful κ times, she needs on average $(nd+1) \cdot 2^\zeta \cdot \kappa$ scratch attempts. We set $\eta_{\mathcal{P}}$ to be the double of this parameter in order to make the probability that he is successful less than κ times exponentially small (to show this fact formally, we rely on the Chernoff–Hoeffding bounds).

Let us now discuss the parameters of the distributed adversary. The two most important ones are the maximal number of slow queries σ and the maximal number ρ of rounds of interaction between the sub-adversaries.

Procedure $\mathsf{scratch}(i, (S_1, \ldots, S_n), Z, W)$

1. Let $Q^0_{n+1} := H(\mathsf{bin}(0) \,||\, \mathsf{bin}(i) \,||\, Z \,||\, W)$
2. For $k = 1$ to d do:
   (a) Let $Q^k_1 := Q^{k-1}_{n+1}$
   (b) For $j = 2$ to $n+1$ let $Q^k_j := H(\mathsf{bin}(j) \,||\, S_j \,||\, Q^k_{j-1})$.
3. Output Q^d_{n+1}.

(a)

Protocol π_{PIK}

The protocol is parameterized with a "moderate hardness" parameter $\zeta \leq \kappa$. It is executed between the $\eta_{\mathcal{P}}$-bounded Prover $\mathcal{P}$ and the $\eta_{\mathcal{V}}$-bounded Verifier $\mathcal{V}$. Both parties take as input $(1^\kappa, S)$, where $S = (S_1, \ldots, S_n)$.

1. The Verifier $\mathcal{V}$ chooses a random *challenge* $Z \in \{0,1\}^\kappa$ and sends it to the Prover $\mathcal{P}$.
2. The Prover $\mathcal{P}$ does the following **parallel search** across different values of $i = \{1, \ldots, \kappa\}$ and *nonces* $W^i \in \{0,1\}^*$:
   (a) Let $Q^i := \mathsf{scratch}(i, S, Z, W^i)$.
   (b) If Q^i starts with ζ zeros then record W^i and **stop the parallel search**.
   The above search is done as long as the Prover did now exhaust her budget for computing hashes (recall the she can make at most $\eta_{\mathcal{P}}$ of them). If this happens before the search is over, then $\mathcal{P}$ outputs $\perp$ to $\mathcal{V}$, who also outputs $\perp$, and then both halt. Otherwise we proceed to the next step.
3. The Prover sends $(W^1, \ldots, W^\kappa)$ to the Verifier.
4. Upon receiving $(W^1, \ldots, W^\kappa)$ the Verifier outputs yes if for *all* $i \in \{1, \ldots, \kappa\}$ it holds that the output of $\widehat{Q}^i$ of $\mathsf{scratch}(i, (S_1, \ldots, S_n), Z, W)$ is such that $H(\widehat{Q}^i)$ starts with ζ zeros. Otherwise, the Verifier outputs no.

(b)

Fig. 1. Proof of Individual Knowledge (PIK) π_{PIK} that satisfies Def. 1 (see Lemma 2). The main procedure is depicted in point (b). It uses a sub-routine $\mathsf{scratch}$ depicted in point (a).

The value of ζ refers to the moderate hardness parameter, i.e., intuitively it should be "moderately hard" to solve around 2^ζ individual $\mathsf{scratch}$ attempts. In the lemma statement we have that $\zeta := \lceil \log_2(\sigma/(\rho \cdot \kappa)) \rceil + 5$, which (using the fact that $d = 2\rho$) means that $2^\zeta \approx 2^6 \cdot \sigma/(d \cdot \kappa)$, or equivalently $2^\zeta \cdot d \cdot \kappa \approx 2^5 \cdot \sigma$. The intuition behind this formula is as follows: every $\mathsf{scratch}$ procedure consists of d mini-rounds, and we require that κ scratches are successful. The probability that a single scratch execution is successful is $2^{-\zeta}$. Therefore we may expect that around $2^\zeta \cdot d \cdot \kappa$ mini-rounds will be executed by the (cheating) Prover. Our knowledge extractors work assuming that a substantial amount of mini-rounds are executed without using the slow queries. This is guaranteed by the assumption that $2^\zeta \cdot d \cdot \kappa$ is substantially larger than σ (in our case: 2^6 times). This is, of course, a very informal argument. In the proof, it will be formalized using the standard Chernoff–Hoeffding bounds.

Finally, let us discuss the α's. Note that the running time $\alpha_{\mathsf{T}} = O(n \cdot \Phi_b)$ means that the extractor runs in time linear in the running time of $\mathcal{A}_b$ multiplied by the length n of the message S. This is ok since we assume that S is not very long (think of it as a cryptographic key), and we can accept the running time of the extractors to be orders of magnitude larger than the one of the adversaries (see Sect. 1.3).Let us now discuss α_{O} and α_{T}. It is illustrative to analyze these values by introducing a parameter β_b, which is equal to the number of hashes that all the adversaries $\mathcal{A}_b$ computed, divided by the number of hashes computed by the honest Prover. Formally: $\beta := \ell/\eta_{\mathcal{P}}$. This, of course, implies that

$$\ell = \beta \cdot (nd + 1) \cdot 2^{\zeta+1} \cdot \kappa. \tag{2}$$

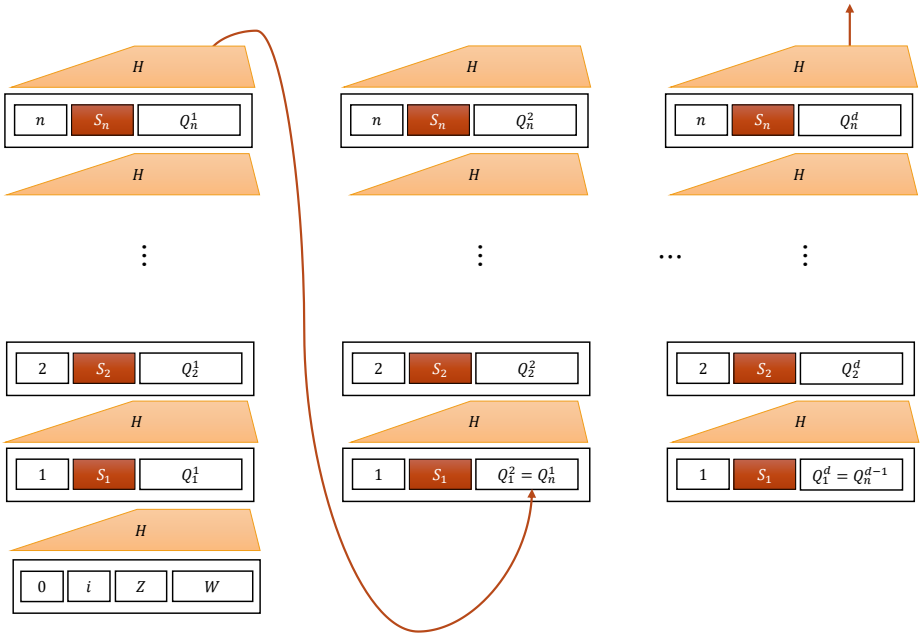

Fig. 2. A diagram representing an execution of $\mathsf{scratch}(i, (S_1, \ldots, S_n), Z, W)$ procedure from Fig. 1.

The best way to think about β is that it defines how much more efficient all the adversarial devices are, compared to the honest execution. Let us now analyze α_{O}, which is the bound on the size of the $\mathcal{E}_b$'s output. From the statement of the Lemma we have

$$\begin{aligned} \alpha_{\mathsf{O}} &:= 2^{14-2\zeta} \cdot \ell^2/(d^2 \cdot \kappa) \\ &= 2^{14-2\zeta} \cdot \left(\beta \cdot (nd+1) \cdot 2^{\zeta+1} \cdot \kappa\right)^2 /(d^2 \cdot \kappa) \qquad (3) \\ &\approx 2^{13} \cdot \beta^2 \cdot \kappa \cdot n, \end{aligned}$$

where Eq. (3) comes from Eq. (2). The fact that in the above formula we have a β^2 factor may look unsatisfactory, but in the current solution this looks unavoidable, since our knowledge extractors do not have any information about the hashes that are computed (other than their outputs), and hence they need to rely on birthday-paradox like phenomena, e.g, to find sequences of hashes that come from a single mini-round.

Finally, let us look at $\alpha_{\mathsf{S}} = O(2^{-\zeta} \cdot \ell/d + \Pi_b)$. Since Π_b is simply the space complexity of $\mathcal{A}_b$, thus the only part that needs to be analyzed is $2^{-\zeta} \cdot \ell/d$. Again applying Eq. (2) we obtain that this value is equal to

$$\begin{aligned} 2^{-\zeta} \cdot \left(\beta \cdot (nd+1) \cdot 2^{\zeta+1} \cdot \kappa/d\right) &= 2 \cdot \beta \cdot (nd+1) \cdot \kappa/d \\ &= O(\beta \cdot n \cdot \kappa). \end{aligned}$$

Concrete parameters. In this paper, we do not attempt to optimize the parameters. Hence for sure, there is much room for improvement. However, let us make a quick analysis of the practical meaning of this result. From the above parameters, the most important one is α_{O}, since it describes the cardinality of the set of "candidate messages S". For $\beta = 100, \kappa = 160, n = 128$ we get that $\alpha_{\mathsf{O}} \approx 8 \cdot 10^{11}$. As we explain in Sect. 3.2, luckily, we can eliminate "false positives" by hashing. Hence, this figure is acceptable in real life since it 10^{11} hashes is less than 1 terrahash.

2.4 Proof of Lemma 2

The proof is divided into three parts. In the Part I, we show that the protocol is complete (due to the lack of space, this part is moved to the Appx. A). In the second one, we prove some facts about the hashes that are computed by a distributed adversary (assuming he convinces the Verifier to accept). In the third one, we construct the knowledge extractors and use the fact from the second part to prove that they have the properties claimed in Lemma 2.

Part II: soundness (analysis of the adversary)
To show the soundness, let $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_a)$ be a (ρ, σ)-distributed adversary. Consider an execution of $\mathcal{A}$ against an honest Verifier on input $(1^\kappa, S)$. We first analyze the behavior of this adversary and then (in Part III) construct the extractors $(\mathcal{E}_1, \ldots, \mathcal{E}_a)$. The goal of Part II of the proof is to show that if $\mathcal{A}$ convinces the Verifier, then there exists a sub-adversary $\mathcal{A}_{\widehat{b}}$ that computed a significant fraction of hashes that are "useful" in the extraction of S. More concretely, what we consider "useful", are the sequences of hashes that match the form of hash sequences in Step 2 on Fig. 1 (a) (for some k), or pictorially: columns on Fig. 2. Moreover, we require that the useful sequences were evaluated in a single round (hence, as we show in Claim 3, they were evaluated by a single sub-adversary $\mathcal{A}_b$), and without any slow queries. Places in the hash-input transcript that are the initial hashes in such hash sequences (i.e., the bottom row of Fig. 2 will be called *tickets*. For a formal definition of the tickets see Def. 4, and the *useful* ones – Def. 5. The bound on the number of useful tickets is given in Claim 6. Informally speaking, the main reason we can prove this bound is that we assume that $\mathcal{A}$ convinces $\mathcal{V}$: in order to do it $\mathcal{A}$ needs to perform a large number of real attempts to solve the puzzles. We call such attempts "serpents"[5] (for a formal definition see Def. 2).

Let us now present the detail of the above proof strategy. Recall that the adversary has access to a σ-bounded oracle Ω_H^{MPC} that she can use to evaluate hash function H. For given execution $((\mathcal{A}_1, \ldots, \mathcal{A}_a) \leftrightarrows \mathcal{V})$ define the following. The *global hash-input transcript IN* is the sequence of all arguments on which the oracle evaluated H. The *global hash-output transcript OUT* is the sequence of all outputs of H that the oracle evaluated. The *local hash-output transcript OUT_b of the sub-adversary $\mathcal{A}_b$* is the sequence of all outputs of H that the oracle computed as a result of the queries from $\mathcal{A}_b$ (and sent to $\mathcal{A}_b$ in response to $\mathcal{A}_b$'s queries). Note that the *local* hash-input transcript was already defined in Sect. 2.1. We assume that the inputs and outputs above are listed in the same order in which H was evaluated on them. We stress that the definitions of the hash-input transcripts are needed only for the analysis, and the extractors take only the hash-output transcripts as input (cf. Fig 3).

For a fixed hash function H, a challenge Z, a message S, and a nonce W let Q_i^k's be the variables in the $\mathsf{scratch}$ procedure (see Fig. 1 or the diagram on Fig. 2) when run on input (i, S, Z, W). For a string $y \in \{0,1\}^*$ define a function Decode that outputs the coordinates" (j, k) that y has on the diagram on Fig. 2 (or $\perp$ if no such coordinates exist), more formally

$$\mathsf{Decode}(i, S, Z, W; y) := \begin{cases} (0,1) & \text{if } y = (\mathsf{bin}(0) \,||\, \mathsf{bin}(i) \,||\, Z \,||\, W) \\ (j,k) & \text{if } y = (\mathsf{bin}(j) \,||\, S_j \,||\, Q_j^k) \\ \perp & \text{otherwise} \end{cases}$$

Note that this function is well-defined, i.e., the decoding of each y is unique. Let $\mathcal{T}$ denote the set of possible non-bottom outputs of Decode, i.e., let $\mathcal{T} := \{(0,1)\} \cup \{1, \ldots, n\} \times \{1, \ldots, d\}$. Fix some challenge Z and a message S. For a hash-input transcript $IN = (IN[1], \ldots, IN[\ell])$ and $i \in \{1, \ldots, \kappa\}$ and $(j,k) \in \mathcal{T}$ and $W \in \{0,1\}^*$ define the following:

$$\mathsf{shot}(IN, W; i, j, k) := \min_{x \in \{1, \ldots, \ell\}} \{x : \ \mathsf{Decode}(i, S, Z, W; IN[x]) = (j, k)\} \tag{4}$$

(assume that $\min_x(\emptyset) := \perp$). In other words: shot points to the first position in the IN transcript where Q_j^k appeared (where Q_j^k is computed by the $\mathsf{scratch}$ procedure from i, S, Z, and W). We now have the following.

[5] This term is chosen since the order of hash computations on the diagram on Fig. 2 resembles a serpent.

Definition 2. *Suppose that for a given (W, i) and all j, k's the sets* $\mathsf{shot}(IN; W; i, j, k)$ *(defined above) are not equal to $\perp$. Then a* serpent *for (W, i) is a sequence defined as follows:* $\mathsf{serpent}(IN, W; i) := \{\mathsf{shot}(IN, W; i, j, k)\}_{j,k}$.

The number of serpents in a given execution will be denoted with C. Below, by saying that some event E happens with "negligible (or overwhelming) probability," we mean that for every poly-time adversary, the probability that E occurs is negligible (resp.: overwhelming) in κ, where the probability is taken over the randomness of the adversary, the Verifier, and the hash function H. We now bound the total number of serpents (not necessarily the successful ones) that the adversary has to compute to have a non-negligible probability of convincing the Verifier. This is done in the following claim, whose proof is moved to Appx. B

Claim 1. Suppose the number C of serpents in the hash-input transcript is at most $\kappa \cdot 2^{\zeta-3}$. Then the probability that $\mathcal{V}$ accepts is negligible.

Recall that our goal is the construct the knowledge extractors $\mathcal{E}_b$ such that for every distributed adversary $\mathcal{A}$ the probability that $\mathcal{V}$ accepts and $S \notin \mathcal{E}_1(OUT_1) \cup \cdots \cup \mathcal{E}_a(OUT_a)$ is negligible. In the rest of this analysis (including all the claims), we assume that $\mathcal{V}$ accepts with non-negligible (as otherwise the statement of the lemma holds trivially). Note that, by Claim 1, if the number C of serpents in the hash-input transcript is at most $\kappa \cdot 2^{\zeta-3}$, then the probability that $\mathcal{V}$ accepts is negligible. Hence the assumption that $\mathcal{V}$ accepts implies that

$$C > \kappa \cdot 2^{\zeta-3}. \tag{5}$$

For every sub-adversary $\mathcal{A}_b$, let ℓ_b denote the number of hashes H that $\mathcal{A}_b$ evaluated (via the Ω_H^{MPC} oracle). Let $\ell := \ell_1 + \cdots + \ell_a$. Fix a global hash input transcript IN. From the remarks above it is clear that if the Verfier accepts, then we can expect that IN contains many traces of the adversary executing the $\mathsf{scratch}$ procedure. Recall, computing every $\mathsf{scratch}$ puzzles takes d "mini-rounds" (i.e.: columns on Fig. 2). Every trace of computation of such a mini-round will be called a "scratch-path". This notion is formally defined below (we define it also for messages $\widehat{S}$ shorter than n since it will be useful later).

Definition 3. *A sequence $(x_1, \ldots, x_{\widehat{n}})$ of elements in $\{1, \ldots, \ell\}$ is called a* scratch-path for a message $\widehat{S} = (\widehat{S}_1, \ldots, \widehat{S}_{\widehat{n}})$ *(for $n \leq \widehat{n}$) if:*

$$IN[x_1] = \mathsf{bin}(1) \,||\, \widehat{S}_1 \,||\, Q \quad \textit{for some } Q \tag{6}$$

and for every $j = 2, \ldots, \widehat{n}$ we have

$$IN[x_j] = \mathsf{bin}(j) \,||\, \widehat{S}_j \,||\, H(IN[x_{j-1}]). \tag{7}$$

The first position of a scratch-path (i.e.: x_1) is called a "ticket". More formally:

Definition 4. *A* ticket for a message $\widehat{S} \in \{0,1\}^n$ *is an index $x \in \{1, \ldots, \ell\}$ such that there exist a scratch path $(x_1, \ldots, x_n)$ for $\widehat{S}$ with $x_1 = x$. We say that x is simply a* ticket *if there exists $\widehat{S}$ such that x is a ticket for $\widehat{S}$.*

Now look again at the input transcript $IN = (IN[1], \ldots, IN[\ell])$. Recall that the computation of $\mathcal{A}$ is divided into ρ rounds of interaction. Since messages in the transcript IN are sorted in the same order they were sent to the oracle, one can split IN into at most ρ sequences of values computed in a single round. Moreover, some (at most σ) hashes are computed using slow queries. Informally speaking, the tickets that "fit into a single round" and are "not computed using a slow query" will be particularly useful for the message extraction in Part III of this proof. This motivates the following definition.

Definition 5. *We say that a ticket x is* useful *if there exists a scratch-path $(x_1, \ldots, x_n)$ for S with $x_1 = x$, such that the following conditions hold:*

1. *all the $IN[x_i]$'s were computed in a single round, and* not *using the slow queries and*
2. *each $IN[x_i]$ above is the first element on IN that has this value, i.e., for every $x < x_i$ we have $IN[x] \neq IN[x_i]$.*

Note that condition 2 above guarantees that $(x_1, \ldots, x_n)$ is unique for a fixed ticket x_1. We now have the following.

Claim 2. With overwhelming probability the total number of useful tickets is at least $C \cdot d/2 - \sigma$.

Proof. It is easy to see that every serpent required d tickets, and unless a collision in H was found, these tickets need to be different for every serpent. Moreover, each such ticket satisfied condition 2, because in the definition of serpent we only look at the *first* occurrences of each x in the IN sequence (cf. Eq. (4)).

We also need to consider condition 1. Since the tickets needed to be computed sequentially, thus at least $d-\rho$ of them are computed in a single round. Hence, with overwhelming probability, each serpent corresponds to $d - \rho$ distinct tickets computed in a single round. This gives us altogether $C \cdot (d - \rho)$ tickets. Removing those that were computed using the slow queries, we obtain at least $C \cdot (d - \rho) - \sigma$ tickets. Using the fact that $d = 2\rho$ we get that this is equal to $C \cdot d/2 - \sigma$. This finishes the proof of the claim. □

For a ticket x_1 of S consider the scratch-path $(x_1, \ldots, x_n)$ for S. If there exists a sub-adversary $\mathcal{A}_b$ such that all the hashes on values $(IN[x_1], \ldots, IN[x_n])$ were computed by $\mathcal{A}_b$ then we say that $\mathcal{A}_b$ *computed the ticket* x). Let X_b denote the number of tickets computed by $\mathcal{A}_b$.

Claim 3. With overwhelming probability every useful ticket x was computed by some $\mathcal{A}_b$.

Proof. Let $\mathcal{A}_b$ be the sub-adversary that evaluated H on $IN[x_n]$. Note that by the assumption that x useful, hash H was evaluated for the first time on all the values $IN[x_1], \ldots, IN[x_{n-1}]$ in the same round as $IN[x_n]$. If $\mathcal{A}_b$ did not evaluate H on some $IN[x_j]$ then the only way she can know remaining $IN[x_{j+1}], \ldots, IN[x_n]$ is to guess $IN[x_j]$ (since this happens in a single round, she cannot get this value from another sub-adversary). Since such a guess is successful with negligible probability, the proof of the claim is finished. □

To conclude this part of the proof: we now know that if $\mathcal{V}$ accepts, then with overwhelming probability, we must have a large number of useful ticket (by Claim 2), and each of them has to be computed entirely by one of the sub-adversaries (by Claim 3). This means that there has to exist some sub-adversary $\mathcal{A}_{\widehat{b}}$ that computed a substantial amount of useful tickets, and hence we can use hope to use the transcript of this $\mathcal{A}_b$ to extract the message S. This is indeed the approach that we will take in Part III of this proof. Informally speaking, each extractor $\mathcal{E}_b$ constructed there will pick up some set of random positions in the hash-output transcripts OUT_b, assume that these positions are winning tickets, and look for matching elements in OUT for different values of candidate messages $\widehat{S}$.

Unfortunately, there is still one technical problem that we need to deal with before we proceed to this part. This problem comes from the fact that a given x can start multiple scratch-paths, and in rare cases, the number of such scratch-paths can be large (we will call such tickets "fat"). This is bad for two reasons. Firstly, it negatively influences the complexity of the extractors. Secondly, it may result in the output of $\mathcal{E}$ being huge (and the size of these sets is one of the parameters we try to optimize). We deal with this problem by showing that the number of fat tickets is bounded. Formally, for every ticket x let $\mathcal{X}(x)$ denote the set containing all scratch paths $(x_1, \ldots, x_{\widehat{n}})$ whose first element is $x_1 = x$ (for all possible messages $\widehat{S}$'s such that $|\widehat{S}| \leq n$), i.e.: $\mathcal{X}(x) = \{X \text{ such that } X \text{ is a scratch-path for some } \widehat{S} \in \{0,1\}^*, \text{ where } |\widehat{S}| \leq n.$

Definition 6. *We say that a ticket x is a* fat *if $|\mathcal{X}(x)| > 4\ell/(C \cdot d)$. Tickets that are not fat are called* slim.

Claim 4. With overwhelming probability, the number of useful fat tickets is at most $C \cdot d/4$.

Proof. Clearly, unless a collision happened, for $x \neq x'$ we have that $\mathcal{X}(x) \cap \mathcal{X}(x')$. Hence, by a counting argument there can be at most $\ell/(4\ell/(C \cdot d/4)) = C \cdot d/4$ tickets that are fat. This finishes the proof of the claim. □

For each $\mathcal{A}_b$ let Γ_b denote the number of useful slim tickets computed by $\mathcal{A}_b$ and let $\Gamma := \Gamma_1 + \cdots + \Gamma_a$. We now have the following.

Claim 5. With overwhelming probability $\Gamma \geq d \cdot \kappa \cdot 2^{\zeta - 6}$.

Proof. By Claim 2, the number of useful tickets is at least $C \cdot d/2 - \sigma$ and by Claim 4, the number of useful fat tickets is at most $C \cdot d/4$. Therefore the number of useful slim tickets is at least

$$C \cdot d/2 - \sigma - C \cdot d/4 = C \cdot d/4 - \sigma \tag{8}$$
$$\geq \kappa \cdot d \cdot 2^{\zeta-5} - \sigma, \tag{9}$$

where Eq. (9) follows from Eq. (5). Recall that in the lemma statement we defined $\zeta := \lceil \log_2(\sigma/(\rho \cdot \kappa)) \rceil + 5$. This implies that $\zeta - 5 \geq \log_2(\sigma/(\rho \cdot \kappa))$, which, in turn means that $\sigma \leq \rho \cdot \kappa \cdot 2^{\zeta-5}$. Substituting $\rho := d/2$ we get $\sigma \leq d \cdot \kappa \cdot 2^{\zeta-6}$, and hence Eq. (9) is at most $\kappa \cdot d \cdot 2^{\zeta-6}$, as required. □

We conclude this part by proving that there has to exist a sub-adversary $\mathcal{A}_b$ that computed a significant number of slim tickets (as a fraction of all the hashes that she evaluated). This this end define $\xi := (d \cdot \kappa \cdot 2^{\zeta-6})/\ell$.

Claim 6. With overwhelming probability there exists $\widehat{b}$ such that $\Gamma_{\widehat{b}}/\ell_{\widehat{b}} \geq \xi$.

Proof. For the sake of contradiction assume that for all b we have that $\Gamma_b/\ell_b < \xi$. This implies that $\Gamma_b < \xi \cdot \ell_b$. Summing up both sides of this inequality over all b's we obtain that $\Gamma < \xi \cdot \ell$, which is equal to $d \cdot \kappa \cdot 2^{\zeta-6}$. This contradicts Claim 5.

Part III: soundness (construction and analysis of the extractors).
We are now ready to construct the extractors $\mathcal{E}_1, \ldots, \mathcal{E}_a$. Each $\mathcal{E}_b$ works in the same way. When lower-bounding the probability that one of them outputs S (see Claim 8 below) we will only consider $\mathcal{E}_{\widehat{b}}$ with $\widehat{b}$ from Claim 6 (since this extractor outputs S with the probability required by the lemma). This is because we cannot assume that a given sub-adversary "knows" that she is the $\mathcal{E}_{\widehat{b}}$, and hence all of them need to run the extraction procedure. However, when upper bounding the output size (in Claim 9) we will consider all the $\mathcal{A}_b$'s. The knowledge extractor is presented in Fig. 3.

Knowledge extractor $\mathcal{E}_b(OUT_b)$

Let $\gamma := \lfloor 2^{6-\zeta} \cdot \ell/d \rfloor$ and $\psi := \lfloor 2^{7-\zeta} \cdot \ell/(d \cdot \kappa) \rfloor$. The algorithm works as follows.

1. Pick random γ values $OUT_b[x_1], \ldots, OUT_b[x_\gamma]$ from the your local hash-output transcript OUT_b.
2. For $i = 1, \ldots, \gamma$ initialize $\mathcal{L}_i^1 := \{(\perp, OUT_b[x_i])\}$.
3. For $j = 2, \ldots, n+1$ do
   (a) For $i = 1, \ldots, \gamma$:
      i. Initialize $\mathcal{L}_i^j := \emptyset$
      ii. For every $(\widehat{S}, \widehat{Q}) \in \mathcal{L}_i^{j-1}$ add $H(\mathsf{bin}(j) \,||\, 0 \,||\, \widehat{Q})$ and $H(\mathsf{bin}(j) \,||\, 1 \,||\, \widehat{Q})$ to set $\mathcal{L}_i^j$.
      iii. If $|\mathcal{L}_i^j| \geq \psi$ then let $\mathcal{L}_i^j := \emptyset$. We also say that $\mathcal{E}_b$ *rejected* i.
   (b) For $i = 1, \ldots, \gamma$ remove from $\mathcal{L}_i^j$ all the elements $(\widehat{S}, \widehat{Q})$ whose first coordinate does *not* appear in the transcript OUT_b.
4. Output the set of elements that are the first coordinates of the elements in sets $\mathcal{L}_{n+1}^j$, i.e., $\{\widehat{S} :$ there exists $\widehat{Q}$ and j such that $(\widehat{S}, \widehat{Q}) \in \mathcal{L}_{n+1}^j\}$.

Fig. 3. Knowledge extractor $\mathcal{E}_b$. See also remarks before Claim 10 for a description how this procedure can be executed without the need of storing the entire OUT_b transcript.

It is easy to see that in Step 3 the extractor essentially computes each set $\mathcal{X}(OUT_b[x_i])$ (where the $\mathcal{X}$'s are defined in on page 15) by computing $\mathcal{L}_i^1, \ldots, \mathcal{L}_i^{n+1}$. The main differences is that the computation is stopped when the set becomes to large ("fat"), in which case, in Step 3(a)iii the extractor "rejects" this

i. More concretely, this happens only if $|\mathcal{L}_i^j| \geq \psi = (4 \cdot \ell)/(d \cdot \kappa \cdot 2^{\zeta-5}) > (4 \cdot \ell)/(C \cdot d)$, where in the last inequality we used the fact that $C > \kappa \cdot 2^{\zeta-3}$ (cf. Eq. (5)). Note that this is precisely the bound that we used in the definition of fat tickets (see Def. 6). Hence, no slim ticket will be rejected in Step 3(a)iii. It is also easy to see that if $\mathcal{E}_b$ did not reject i in Step 3(a)iii and $OUT_b[x_i]$ is a useful ticket, then $S \in \mathcal{E}_b(OUT_b)$. We hence get the following.

Claim 7. If x_i is a slim useful ticket then $S \in \mathcal{E}_b(OUT_b)$

We can now show the following claim.

Claim 8. With overwhelming probability $S \in \mathcal{E}_{\widehat{b}}(\mathcal{Q}_{\widehat{b}})$.

Proof. By Claim 7 it suffices to show that the probability that out of randomly-chosen γ values $x_1, \ldots, x_\gamma$ in Step 1 at least one is a slim useful ticket with overwhelming probability. Observe that $\gamma = \left\lfloor \frac{2^{6-\zeta} \cdot \ell}{d} \right\rfloor$, which is equal to $\left\lfloor \frac{\kappa}{\xi} \right\rfloor$ (cf. the definition of ξ). By Claim 6 the fraction $\Gamma_{\widehat{b}}/\ell_{\widehat{b}}$ of indices that are useful slim tickets in the transcript of $\mathcal{A}_{\widehat{b}}$ is at least ξ. We hence get that the probability that at least one of $x_1, \ldots, x_\gamma$ is a slim useful ticket is at least equal to $\Gamma_{\widehat{b}}/\ell_{\widehat{b}} \geq \xi$. The probability that this happens for at least one t_j is hence at least $1 - (1-\xi)^\gamma \geq 1 - (1-\xi)^{\lfloor \kappa/\xi \rfloor} \geq 1 - \mathsf{negl}(\kappa)$ where e is the Euler's number and we use the fact that for $\xi \in (0,1)$ we have $(1-\xi)^{1/\xi} \leq e^{-1}$. This finishes the proof of this claim. □

Note that Claim 8 implies that Eq. (1) of the PIK definition (Def. 1) holds. What remains is to show that the extraction efficiency parameters ($\alpha_{\mathsf{O}}, \alpha_{\mathsf{T}}$, and α_{S}) are as required by the lemma statement. This is done in Claim 9 (for α_{O}), and in Claim 10 (for α_{T} and α_{S}).

Claim 9. For every b the size α_{O} of the output of $\mathcal{E}_b(\mathcal{Q}_b)$ is at most $\alpha_{\mathsf{O}} := (2^{14-2\zeta} \cdot \ell^2)/(d^2 \cdot \kappa)$.

Proof. Clearly, the cardinality of each $\mathcal{L}_i^{n+1}$ is at most $2 \cdot \psi$ (as otherwise i would be rejected in Step 3(a)iii). Since we have γ such sets, thus the size of the output of $\mathcal{E}_b(\mathcal{Q}_b)$ is at most $\gamma \cdot 2 \cdot \psi \leq 2 \cdot \left\lfloor 2^{6-\zeta} \cdot \ell/d \right\rfloor \cdot \left\lfloor 2^{7-\zeta} \cdot \ell/(d \cdot \kappa) \right\rfloor \leq 2^{14-2\zeta} \cdot \ell^2/(d^2 \cdot \kappa)$ and the claim is proven. □

For bounding α_{S} and α_{T} observe that the most space-consuming part of $\mathcal{E}_b$ (as described on Fig. 3) is storing the entire hash-output transcript OUT_b, since typically this number is large. Luckily we can exploit the fact that OUT_b can be generated quickly (usually: $\mathcal{A}_b$ will have to generate it in a few seconds). Recall also that $\mathcal{E}_b$ is generated locally by $\mathcal{A}_b$ as a function of $\mathcal{A}_b$'s internal randomness and the inputs that $\mathcal{A}_b$ received (which can be upper-bounded by the amount of space used by $\mathcal{A}_b$). Hence, if we store this information, we can "regenerate" the transcript OUT_b from it "on the fly" whenever it is needed. This happens $n+1$ times: once in Step 1 and n times in Step 3b. Further, to make the check in Step 3b more efficient, we can store the values $\mathcal{L}_i^j$ in a hash table. By using standard techniques such as the Cuckoo Hashing [27] checking if a given $OUT_b(x)$ appears in $\mathcal{L}_i^j$ (as the first coordinate of the $(\widehat{S}, \widehat{Q})$ pair) takes constant time. The preprocessing in Cuckoo Hashing takes $O(|\mathcal{L}_i^j|)$ time and space, which is $O(|OUT_b|)$ (since $|\mathcal{L}_i^j| \leq |OUT_b|$). We, therefore, get the following.

Claim 10. Let $\mathsf{T}_{\mathcal{A}_b}$ be the running time of $\mathcal{A}_b$ and let ℓ be the number of hashes H that $\mathcal{A}$ evaluated. Then, the execution of algorithm $\mathcal{E}_b$ from Fig. 3, with the optimizations described above, takes time $O(n \cdot \mathsf{T}_A)$ and space $O(2^{-\zeta} \cdot \ell/d)$ plus space needed to store all the messages that $\mathcal{A}_{\widehat{b}}$ received.

Proof. Follows from the remarks above, and the fact that $|\mathcal{L}_i^j|$ is at most $2 \cdot \psi_d = O(2^{-\zeta} \cdot \ell/(d \cdot \kappa))$ (when measured in blocks of length $O(\kappa)$, and hence $|\mathcal{L}_i^j|$ is at most $O(2^{-\zeta} \cdot \ell/d)$ when measured in bits.

The above two claims conclude the proof.

3 Extensions

This section describes possible extensions of the basic PIK protocol from Sect. 2. We do it in a rather informal way, leaving the formalization of these ideas for future work. Note that formal modeling of attacks in the network settings is often highly non-trivial, as one needs to consider different attack scenarios (e.g., man-in-the middle, replay attacks, etc.), and typically requires a detailed description of the attack model (including modeling the network, and the environment). Hence it would not fit into this paper within the page limit.

3.1 PIK as building block

As mentioned in Sect. 1, PIK is a primitive that will usually not be used in a "standalone" way but rather as a part of a more extensive system. The main reason for this is that the messages sent by Prover may provide some information about S. Indeed, the definition of PIK does not mention any privacy guarantees for the Prover, neither against an external attacker nor against the Verifier (who actually, in our settings from Def. 1 knows S entirely). In this section, we describe two extensions of PIK that address this problem. The first one ("Encrypted and Authenticated PIK") answers the problem of security against an external adversary, and the second one provides a version of PIK ("zero-knowledge PIK") that works against the Verifier that does *not* know S. This version of PIK does not reveal any information about S to the Verifier (or to any other party).

Encrypted and Authenticated PIK (eaPIK). *Encrypted and Authenticated PIK (eaPIK)* is a version of PIK where the communication between the Prover and the Verifier is secured using a symmetric key K. Let us describe this solution as a general transformation of any PIK protocol π protocol. The main idea is quite straightforward: we let every message[6] in the protocol π be encrypted and authenticated with a fresh key K that is shared between the Prover and the Verifier, sampled independently from S. Note that in our applications (see Sect. 4), secret S also plays a role of a key that the Client (playing the role of the Prover) and the Server (the Verifer) agreed on during a setup phase. The main idea is that the possession of S unlocks full access to the Server's services. Note that, since we anyway have a setup phase (for establishing S), thus assuming that in this phase we also generate K is very mild. To summarize: the Client and the Server share the following:

- key K for encrypting messages during the execution of the PIK protocol and
- secret S that is used as the input of both parties in PIK.

Of course, as long as the Prover is honest (and hence: she stores S on a single machine), she easily convinces the Verifier that she knows S. On the other hand, by the PIK properties, a dishonest Prover cannot distribute S among different sub-adversaries. It is important to note that the PIK protocol only guarantees that S cannot be distributed, and K is not protected in the same way. Hence, in any potential application accessing the server should require the knowledge of S only, and it should not be assumed that K is stored on a single machine.

Deriving S from K. One promising approach is to derive S from K. At first sight, it may look tempting to let K be equal to S (possibly truncated). This solution, unfortunately, would not work for the following reasons: (a) in some applications, message S is not guaranteed to be random, (b) standard security definitions for encryption do not guarantee that the entire key remains hidden, and consequently, the encrypted messages could reveal some information about S, and (c) encrypting messages that depend on a key K with the same key creates problems that are known as *key-dependent message security* (see, e.g., [25]).

[6] It is easy to see that in case of our protocol π_{PIK} (see Fig. 1) the only message that may contain sensitive information about S is $(W^1, \ldots, W^\kappa)$ sent by the Prover to the Verifier in Step 3. Hence, it is enough if only this message is encrypted.

These problems can be solved by letting $K := H''(S)$ (where H'' is some hash function different from the one used elsewhere in the construction) and using key-dependent authenticated encryption. Informally, this solution works assuming that S has a sufficiently high-entropy (which is the case in the applications from Sect. 4). We leave analyzing this as future work.

ZK-Proofs of Individual Knowledge. Let us now describe the solution in the settings where S is known only to the Prover. In this case, PIK makes sense only if some publicly-available information on S is known. We model this in the following standard way. Suppose L is an NP-language characterized by some NP-relation $\varphi : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$. The Prover holds $S \in \{0,1\}^*$, while the Verifier holds an NP-witness $\omega \in \{0,1\}^*$. A natural example of L is the language of all secret keys in some public-key encryption scheme, with ω being the public key such that $\varphi(S,\omega) = 1$ if and only if S is the secret key that corresponds to ω. The goal of the Prover is to convince the Verifier that she knows S such that $\varphi(S,\omega) = 1$, and that this S is stored on an individual machine. Moreover, this should be done in zero-knowledge, i.e., without revealing additional information to the Verifier about S. We call a protocol that satisfies these requirements a *ZK-Proof of Individual Knowledge (zkPIK)* for relation φ.

Let us now outline how to transform a public-coin PIK protocol π into a zkPIK, where by "public-coin PIK" we mean protocols where the messages of the Verifier is drawn at random, independently from S. It is easy to see that or π_{PIK} protocol (see Fig. 1) satisfies is public-coin, since the only messages that are sent by the Verifier is the random challenge Z in Step 1. Our protocol works as follows. The Prover and the Verifier execute π with the following modification: the Prover, instead of sending her messages in clear, simply commits to them using a cryptographic commitment scheme [11], and later proves in zero-knowledge [21] that she committed to messages that are such that the Verifier in protocol π would accept. Note that the zero-knowledge proof can be executed after the message exchange is finished, and hence time needed to execute it does not influence the time bounds of the original PIK protocol. Clearly, this proof can also be executed in a non-interactive way [10], e.g., using one of the Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zkSNARK) schemes (see, e.g., [9]). Note that this solution assumes a generic use of zero-knowledge protocols. We leave it as an open problem to develop more efficient protocols that do not rely on such generic techniques.

3.2 Reducing the number of candidate messages.

Recall that one of the parameters in PIK is the maximal size α_{O} of the set of "candidate secrets $\widehat{S}$", that besides the real value S contains a lot of "false positives". Moreover in our construction α_{O} can be quite large (see the "Concrete parameters" section on page 12). A simple technique for eliminating the false positives is to let the Verifier publish a hash $h = H'(S)$ (where H' is some hash function different from the one used in constructing PIK). Once h is known, everyone (including the Prover) can check for every candidate $\widehat{S}$ if $H(\mathsf{H}) = h$, and if it does not hold, eliminate this H from the candidate set. It is easy to see that $\widehat{S} = S$ passes this test, while (assuming no collisions in H are found) all $\widehat{S}$'s such that $\widehat{S} \neq S$ get eliminated. This solution can be proven secure in the random oracle model as long as the entropy of S (from the point of view of an external adversary) is large enough to guarantee that S cannot be guessed by the adversary. This is the case, e.g., if S is a uniform random key, like in the applications that we describe in Sect. 4.

Let us also discuss how h can be published. One option is to let h be sent to the Verifier by the Prover during the execution of the protocol. In the case of the eaPIK protocol, h would be encrypted by key K. Note that we do not even need the assumption that S has high entropy since it remains hidden from the external attacker. In the case of zkPIK, h could just be a part of the public key.

4 Conclusion

In this paper, we initiated the formal study of individual cryptography. Besides being interesting from a theoretical point of view, it can be viewed as a technique for preventing the misuse of online services using

MPC or TEE techniques. Our approach is primarily theoretical, and it would be exciting to search for particular computational problems that are MPC- or TEE-hard in practice (one good candidate would be functions that require many memory lookups), and to improve the practical parameters that we achieve. This would be, in a sense, a complementary effort to the search for "MPC-friendly primitives" (see, e.g., [22]).

References

[1] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro. "Scrypt Is Maximally Memory-Hard". In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*. 2017.

[2] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. "Secure Multiparty Computations on Bitcoin". In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. 2014.

[3] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A. Sadeghi, G. Scerri, and B. Warinschi. "Secure Multiparty Computation from SGX". In: *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*. 2017.

[4] C. Baum, B. David, and R. Dowsley. "Insured MPC: Efficient Secure Computation with Financial Penalties". In: *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*. 2020.

[5] M. Bellare and O. Goldreich. "On Defining Proofs of Knowledge". In: *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*. 1992.

[6] M. Bellare and P. Rogaway. "Optimal Asymmetric Encryption". In: *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*. 1994.

[7] M. Bellare and P. Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*. 1993.

[8] I. Bentov and R. Kumaresan. "How to Use Bitcoin to Design Fair Protocols". In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*. 2014.

[9] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer. "The Hunting of the SNARK". In: *J. Cryptol.* 4 (2017).

[10] M. Blum, P. Feldman, and S. Micali. "Non-Interactive Zero-Knowledge and Its Applications (Extended Abstract)". In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. 1988.

[11] G. Brassard, D. Chaum, and C. Crépeau. "Minimum Disclosure Proofs of Knowledge". In: *J. Comput. Syst. Sci.* 2 (1988).

[12] N. Chandran, V. Goyal, R. Moriarty, and R. Ostrovsky. "Position-Based Cryptography". In: *SIAM J. Comput.* 4 (2014).

[13] B. Chor, A. Fiat, M. Naor, and B. Pinkas. "Tracing traitors". In: *IEEE Trans. Inf. Theory* 3 (2000).

[14] I. Damgård. "Towards Practical Public Key Systems Secure Against Chosen Ciphertext Attacks". In: *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*. 1991.

[15] D. P. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009.

[16] C. Dwork, M. Naor, and H. Wee. "Pebbling and Proofs of Work". In: *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*. 2005.

[17] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. "Proofs of Space". In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*. 2015.

[18] S. Dziembowski, T. Kazana, and D. Wichs. "One-Time Computable Self-erasing Functions". In: *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*. 2011.

[19] S. Fei, Z. Yan, W. Ding, and H. Xie. "Security Vulnerabilities of SGX and Countermeasures: A Survey". In: *ACM Comput. Surv.* 6 (2021).

[20] J. A. Garay, A. Kiayias, and N. Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*. 2015.

[21] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof Systems". In: *SIAM J. Comput.* 1 (1989).

[22] L. Grassi, C. Rechberger, D. Rotaru, P. Scholl, and N. P. Smart. "MPC-Friendly Symmetric Key Primitives". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016.

[23] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[24] A. Kiayias and Q. Tang. "How to keep a secret: leakage deterring public-key cryptosystems". In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 2013.

[25] T. Malkin, I. Teranishi, and M. Yung. "Key dependent message security: recent results and applications". In: *First ACM Conference on Data and Application Security and Privacy, CODASPY 2011, San Antonio, TX, USA, February 21-23, 2011, Proceedings*. 2011.

[26] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. "A New Approach to Practical Active-Secure Two-Party Computation". In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. 2012.

[27] R. Pagh and F. F. Rodler. "Cuckoo hashing". In: *J. Algorithms* 2 (2004).

[28] I. Puddu, D. Lain, M. Schneider, E. Tretiakova, S. Matetic, and S. Capkun. "TEEvil: Identity Lease via Trusted Execution Environments". In: *CoRR* (2019).

[29] T. Ruffing, A. Kate, and D. Schröder. "Liar, Liar, Coins on Fire!: Penalizing Equivocation By Loss of Bitcoins". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 2015.

[30] Wikipedia contributors. *Trusted execution environment — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-October-2022]. 2022.

[31] A. L. Young and M. Yung. "Cryptovirology: Extortion-Based Security Threats and Countermeasures". In: *1996 IEEE Symposium on Security and Privacy, May 6-8, 1996, Oakland, CA, USA*. 1996.

Supplementary Material

Summary of the notation used in the definition	
$S = (S_1, \ldots, S_n)$	– the secret message
$\mathcal{A}_1, \ldots, \mathcal{A}_a$	– the sub-adversaries
a	– the number of sub-adversaries
$\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_a)$	– the distributed adversary
ρ	– the number of rounds
1^κ	– the computational security parameter
ζ	– the moderate hardness parameter

Summary of the notation used in the protocol	
Z	– the challenge
W	– the nonce
$d\ (= 2\rho)$	– the circuit depth
Q_j^k's	– intermediate variables.
$\nu_{\mathcal{P}} := (nd+1) \cdot 2^{\zeta+1} \cdot \kappa$	– the number of hashes evaluated by the honest Prover
$\nu_{\mathcal{V}} := (nd+1) \cdot \kappa$	– the number of hashes evaluated by the honest Verifier

Summary of the most important notation used in the proof	
IN	– global hash-input transcript
IN_b	– global hash-input transcript of the sub-adversary $\mathcal{A}_b$
OUT	– global hash-output transcript
OUT_b	– local hash-output transcript of the sub-adversary $\mathcal{A}_b$
$(\mathcal{T}, \prec)$	– lexicographically-ordered set of "coordinates" on the diagram on Fig. 2
Decode	– function used to compute "coordinates" of a string
ℓ_b	– the number of hashes H a sub-adversary $\mathcal{A}_b$ evaluated
$\ell = \ell_1 + \cdots + \ell_a$	– the number of hashes H evaluated by all the sub-adversaries
shot	– see Eq. (4)
X_b	– the number of tickets computed by a sub-adversary $\mathcal{A}_b$
serpent	– see Def. 2
C	– the number of serpents

Fig. 4. Summary of the notation.

A Completeness proof of PIK

Let us start with proving completeness. This argument is quite standard. From the construction of the protocol it is clear that the computation of $\mathcal{V}$ takes $\eta_{\mathcal{V}} = (nd+1) \cdot \kappa$ hash evaluations ($nd+1$ hashes per each $i \in \{1, \ldots, \kappa\}$). Hence, let us focus on $\mathcal{P}$. We first upper-bound the probability that the protocol halts in Step 2 due to the fact that the budget for H computations was exhausted. Note that this budget allows the Prover to evaluate scratch $\lfloor \eta_{\mathcal{P}}/(nd+1) \rfloor = 2^{\zeta+1} \cdot \kappa$ times (since each scratch execution takes $nd+1$ hash evaluations). For $i = 1, \ldots, 2^{\zeta+1} \cdot \kappa$ let $U_i \in \{0,1\}$ be equal to 1 if and only if the ith scratch execution was successful, i.e., it produced an output Q such that $H(Q)$ starts with κ zeros. Set $U = U_1 + \cdots + U_{2^{\zeta+1} \cdot \kappa}$.

For simplicity of the analysis assume that π_{PIK} does not stop once all the W^j's are found. Clearly, the U_i's are independent. Obviously each $\Pr[U_i = 1] = 2^{-\zeta}$ (the probability that a random string starts with ζ zeros) and hence each $\mathbb{E}[U_i] = 2^{-\zeta}$. Therefore: $\mathbb{E}[U] = 2^{\zeta+1} \cdot \kappa \cdot 2^{-\zeta} = 2 \cdot \kappa$. We can now use the Chernoff–Hoeffding bound from Lemma 1 (point (2)) with $\epsilon = 1/2$, obtaining

$$\Pr\left[U < \frac{1}{2} \cdot 2 \cdot \kappa\right] \leq \exp\left(-\frac{1}{4} \cdot 2 \cdot \kappa/2\right),$$

which means that $\Pr[U < \kappa] \leq \mathsf{negl}(\kappa)$. Therefore with overwhelming probability the Prover finds κ values Q such that $H(Q)$ starts with ζ zeros without exhausting her hash budget, and hence she does not output $\perp$. It is also easy to see that the Verifer always accepts in such a case (since she just repeats the same $\mathsf{scratch}$ computation as the Prover for the W^i's that she received in Step 4).

B Proof of Claim 1

B.1 Auxiliary claims

We first show that with overwhelming probability different serpents are disjoint. To be more formal, let E_{11} denote the event that this is not the case, i.e., let

$$\begin{aligned}\mathsf{E}_{11} = \text{“there exists } &(W, i) \neq (W', i') \text{ such that} \\ &\mathsf{serpent}(IN, W; i) \cap \mathsf{serpent}(IN, W'; i') \neq \emptyset\text{.”}\end{aligned}$$

Claim 11. The probability of E_1 is negligible.

Proof. Recall that the Decode function decodes the inputs uniquely for (i, S, Z, W), and we fixed (Z, S). Hence for E_1 *not* to hold, we need to find (i_0, W_0) and (i_1, W_1) such that $(i_0, W_0) \neq (i_1, W_1)$ and $Q^k_{i,0} \neq Q^k_{i,1}$, where $Q^k_{i,0}$ and $Q^k_{i,1}$ are the variables that appear in the computation of $\mathsf{scratch}$ on inputs (i_0, S, Z, W_0) and (i_1, S, Z, W_1) respectively. It is easy to see that this only happens if a collision in H is found, and hence the probability of this event is negligible. This finishes the proof of the claim. □

Below, let "$\prec$" denote the *strict* lexicographic ordering of the set $\mathcal{T}$, i.e., ordering such that $(j, k) \prec (j', k')$ if $j < j'$ or ($j = j'$ and $k < k'$). Note that in the honest execution of the $\mathsf{scratch}$ procedure this order corresponds to the order in which H is evaluated on inputs y (if we look at the values (j, k) that they decode do). Let E_2 be the event that this did not happen in a *dishonest* execution. Namely define:

$$\begin{aligned}\mathsf{E}_2 := \text{“for some } &\mathsf{serpent}(IN, W; i) = \{x_{j,k}\}_{j,k} \text{ there exists} \\ &(i, j) \prec (i', j') \text{ such that } x_{j,k} \geq x_{i',j'}\text{”}.\end{aligned}$$

Claim 12. The probability of E_2 is negligible.

Proof. Note that we defined each $\mathsf{shot}(IN, W; i, j, k)$ to be the minimal x such that $IN[x]$ decodes to (j, k). Suppose $x_{j,k} \geq x_{i',j'}$ and $(i, j) \prec (i', j')$. Obviously, from the uniqueness o the decoding $x_{j,k}$ cannot be equal to $x_{i',j'}$'. Hence we can assume that $x_{j,k} > x_{i',j'}$. But this means that $Q^{k'}_{j'}$ (that depends on Q^k_j) was computed before Q^k_j. Since we model H as a random oracle, this can only happen with negligible probability. This finishes the proof of the claim. □

Note that the above claim implies that every serpent contains $nd+1$ distinct elements. Recall that $\mathcal{V}$ accepts if it receives $W^1, \ldots, W^\kappa$ such that for all i we have that $H(\widehat{Q}^i)$ starts with ζ zeros (cf. Step 4 on Fig. 1 (b)), where every $\widehat{Q}^i$ is computed from S, Z, and W^i in the $\mathsf{scratch}(i, S, Z, W)$ procedure (see Fig. 1 (a)). For a hash-input transcript IN and $\mathsf{serpent}(IN, W, i) = \{x_{j,k}\}_{j,k}$ take the last element $IN[x_{\kappa,n}]$ that forms this serpent. We call this serpent *successful* if the hash $H(n \,||\, S_n \,||\, IN[x_{\kappa,n}])$ starts with ζ zeros. Define

$$\mathsf{E}_3 := \text{“The number of successful serpents is less than } \kappa/2 \text{ and } \mathcal{V} \text{ accepted.”}$$

In other words: a serpent is successful if its existence in the transcript means that the adversary found a nonce W^i (for Z and S) that convinces the Verifier. We now have the following fact, that, informally speaking, states a very intuitive fact that if the probability that number of successful serpents is less than $\kappa/2$ is and the Verifier accepted is negligible.

Claim 13. The probability of E_3 is negligible.

Proof. It is easy to see that the probability that a given W^i is such that the corresponding $H(\widehat{Q}^i)$ starts with ζ zeros and there was no successful serpent for (W, i) is at most $2^{-\zeta} + \mathsf{negl}(\kappa)$. This is because if the adversary did not evaluate on all inputs from some successful shot (W, i), then there needs to exists some value (j, k) such that

$$\mathsf{shot}(IN, W; i, j, k) = \perp,$$

which means that H was never evaluated on one of the inputs that are used in the computation of $H(\widehat{Q}^i)$. Since all the proceeding input (according to the "$\prec$" ordering) are dependent on the outputs of this hash (and the hash has length κ), thus the output of $H(\widehat{Q}^i)$ is uniformly random, and the probability that it starts with ζ zeros is $2^{-\zeta} + \mathsf{negl}(\kappa)$. Clearly, the probability that this happens at least $\kappa/2$ times is then at most $(2^{-\zeta} + \mathsf{negl}(\kappa))^{\kappa/2} \leq \mathsf{negl}(\kappa)$. □

B.2 Proof

After proving the above claims, we are now ready to prove Claim 1. Assume that none of the events $\mathsf{E}_1, \mathsf{E}_2$ and E_3 occurred. We can make this assumption, since by the claims above these events have negligible probabilities. The probability that a given serpent is successful is equal to $2^{-\zeta}$, i.e., the probability that an output of H starts with ζ zeros (note that here we use the assumption that E_2 occurred, i.e., the adversaries have to *first* query the oracle H on all the inputs from a given serpent, and *then* they learn if the it was successful).

For each $i \in \{1, \ldots, C\}$ let $S_i \in \{0, 1\}$ be equal to 1 if and only if the ith serpent was successful. Since we assumed that E_1 occurred, then the x's that they are composed of are pairwise disjoint. Therefore the S_i's are independent. Let $U = U_1 + \cdots U_C$. Since $\Pr[U_i = 1] = 2^{-\zeta}$, thus $\mathbb{E}[U] = 2^{-\zeta} \cdot C$. Since we assumed that E_3 did not occur, $\mathcal{V}$ accepts only if there were at least $\kappa/2$ successful serpents, which means that $U > \kappa/2$. We now have:

$$\begin{aligned}
\Pr[U > \kappa/2] &= \Pr\left[U > 2^{-\zeta} \cdot C + (\kappa/2 - 2^{-\zeta} \cdot C)\right] && (10)\\
&\leq \exp\left(-2\left(\kappa/2 - 2^{-\zeta} \cdot C\right)^2 / C\right) && (11)\\
&= \exp\left(-\frac{\kappa^2}{2C} + \kappa \cdot 2^{1-\zeta} - C \cdot 2^{1-2\zeta}\right) && (12)\\
&\leq \exp\left(-\frac{\kappa^2}{2C} + \kappa \cdot 2^{1-\zeta}\right) && (13)\\
&\leq \exp\left(-\frac{\kappa^2}{2 \cdot \kappa \cdot 2^{\zeta-3}} + \kappa \cdot 2^{1-\zeta}\right) && (14)\\
&= \exp\left(-\kappa \cdot 2^{2-\zeta} + \kappa \cdot 2^{1-\zeta}\right) && (15)\\
&= \exp\left(-\kappa \cdot 2^{1-\zeta}\right) && (16)\\
&\leq \mathsf{negl}(\kappa) && (17)
\end{aligned}$$

Above in Eq. (11) we used the Chernoff–Hoeffding bound from Lemma 1 (point (1)) with $\delta = \kappa/2 - 2^{-\zeta} \cdot C$ and in Eq. (14) – the assumption that $C \leq \kappa \cdot 2^{\zeta-3}$. This finishes the proof of Claim 1.