# Resettably Sound Zero-Knowledge Arguments from OWFs - The (Semi) Black-Box Way

Rafail Ostrovsky[1,⋆], Alessandra Scafuro[2,⋆⋆], and Muthuramakrishnan Venkitasubramanian[3]

[1] UCLA, USA
[2] Boston University and Northeastern University, USA
[3] University of Rochester, USA

**Abstract.** We construct a constant round resettably-sound zero knowledge argument of knowledge based on black-box use of any one-way function. Resettable-soundness was introduced by Barak, Goldreich, Goldwasser and Lindell [FOCS 01] and is a strengthening of the soundness requirement in interactive proofs demanding that soundness should hold even if the malicious prover is allowed to "reset" and "restart" the verifier. In their work they show that resettably-sound ZK arguments require non-black-box simulation techniques, and also provide the first construction based on the breakthrough simulation technique of Barak [FOCS 01]. All known implementations of Barak's non-black-box technique required non-black-box use of a collision-resistance hash-function (CRHF).

Very recently, Goyal, Ostrovsky, Scafuro and Visconti [STOC 14] showed an implementation of Barak's technique that needs only black-box access to a collision-resistant hash-function while still having a non-black-box simulator. (Such a construction is referred to as *semi black-box*.) Plugging this implementation in the compiler due to Barak et al. yields the first resettably-sound ZK arguments based on black-box use of CRHFs.

However, from the work of Chung, Pass and Seth [STOC 13] and Bitansky and Paneth [STOC 13], we know that resettably-sound ZK arguments can be constructed from non-black-box use of any one-way function (OWF), which is the *minimal* assumption for ZK arguments.

Hence, a natural question is whether it is possible to construct resettably-sound zero-knowledge arguments from black-box use of any OWF only. In this work we provide a positive answer to this question thus closing the gap between black-box and non-black-box constructions for resettably-sound ZK arguments.

---

⋆ Work supported in part by NSF grants 09165174, 1065276, 1118126 and 1136174, US-Israel BSF grant 2008411, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, and LockheedMartin Corporation Research Award; and DARPA under Contract N00014 -11 -1-0392. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

⋆⋆ Work done while working at UCLA.

Y. Dodis and J.B. Nielsen (Eds.): TCC 2015, Part I, LNCS 9014, pp. 345–374, 2015.
© International Association for Cryptologic Research 2015

## 1 Introduction

Zero-knowledge (ZK) proofs [13] allow a prover to convince a verifier of the validity of a mathematical statement of the form "$x \in L$" without revealing any additional knowledge to the verifier besides the fact that the theorem is true. This requirement is formalized using a simulation paradigm: for every malicious verifier there exists a simulator that having a "special" access to the verifier (special in the sense that the access granted to the simulator is not granted to the prover) but no witness, is able to reproduce the view that the verifier would obtain interacting with an honest prover (who knows the witness). The simulator has two special accesses to the verifier: black-box access, it has the power of resetting the verifier during the simulation; non-black-box access, it obtains the actual code of the verifier. While providing no additional knowledge, the proof must also be sound, i.e. no malicious prover should be able to convince a verifier of a false statement.

In this work we consider a stronger soundness requirement where the prover should not be able to convince the verifier even when having the power of resetting the verifier's machine (namely, having black-box access to the verifier). This notion of soundness, referred to as *resettable soundness*, was first introduced by Barak, Goldwasser, Goldreich and Lindell (BGGL) in [3], and is particularly relevant for cryptographic protocols being executed on embedded devices such as smart cards. Barak et al. in [3] prove that, unless $\mathbf{NP} \subseteq \mathsf{BPP}$, interactive proofs for **NP** cannot admit a black-box zero knowledge simulator and be resettably-sound at same time. (Indeed, a resetting prover has the same special access to the verifier as a black-box simulator and it can therefore break soundness just by running the simulator's strategy.) Then, they provide the first resettably-sound zero-knowledge arguments for NP based on the non-black-box zero-knowledge protocol of [1] and on the existence of collision-resistant hash-functions (CRHFs). Recently, Chung, Pass and Seth (CPS) [9] showed that the minimal assumption for non-black-box zero-knowledge is the existence of one-way functions (OWFs). In their work they provide a new way of implementing Barak's non-black-box simulation strategy which requires only OWFs. Independently, Bitansky and Paneth [5] also showed that OWFs are sufficient by using a completely new approach based on the impossibility of approximate obfuscation.

Common to all the above constructions [3,9,5], beside the need of non-black-box simulation, is the need of non-black-box use of the underlying cryptographic primitives. Before proceeding, let us explain the meaning of black-box versus non-black-box use of a cryptographic primitive. A protocol makes black-box use of a cryptographic primitive if it only needs to access the input/output interface of the primitive. On the other hand, a protocol that relies on the knowledge of the implementation (e.g., the circuit) of the primitive is said to rely on the underlying primitive in a non-black-box way. A long line of work [19,17,28,6,31,14,22,15] starting from the seminal work of Impagliazzo and Rudich [18] aimed to understand the power of the non-black-box access versus the black-box access to a cryptographic primitive. Besides strong theoretical motivation, a practical reason is related to efficiency. Typically, non-black-box constructions are inefficient

and as such, non-black-box constructions are used merely to demonstrate "feasibility" results. A first step towards making these constructions efficient is to obtain a construction that makes only black-box use of the underlying primitives.

In the resettable setting non-black-box simulation is necessary. In this work we are interested in understanding if non-black-box use of the underlying primitive is necessary as well. Very recently, [16] constructed a public-coin ZK argument of knowledge based on CRHFs in a black-box manner. They provided a non black-box simulator but a black-box construction based on CRHFs. Such a reduction is referred to as a *semi black-box* construction (see [29] for more on different notions of reductions). By applying the [3] transformation, their protocol yields the first (semi) black-box construction of a resettably-sound ZK argument that relies on CRHFs. In this paper, we address the following open question:

> *Can we construct resettably-sound ZK arguments under the minimal assumption of the existence of a OWF where the OWF is used in a black-box way?*

### 1.1 Our Results

We resolve this question positively. Formally, we prove the following theorem.

**Theorem 1 (Informal).** *There exists a (semi) black-box construction of an O(1)-round resettably-sound zero-knowledge argument of knowledge for every language in NP based on one-way functions.*

It might seem that achieving such result is a matter of combining techniques from [16], which provides a "black-box" implementation of Barak's non-black-box simulation and [9], which provides an implementation of Barak's technique based on OWFs. However, it turns out that the two works have conflicting demands on the use of the underlying primitive which make the two techniques "incompatible".

More specifically, the construction presented in [16] crucially relies on the fact that a collision-resistance hash-function is publicly available. Namely, in [16] the prover (and the simulator) should be able to evaluate the hash function *on its own* on any message of its choice at any point of the protocol execution. In contrast, the protocol proposed in [9] replaces the hash function with digital signatures (that can be constructed from one-way functions), and requires that the signature key is hidden from the prover: the only way the prover can obtain a signature is through a "signature slot". Consequently in [9], in contrast with [16], the prover cannot compute signatures on its own, cannot obtain signatures at any point of the protocol (but only in the signature slot), and cannot obtain an arbitrary number of signatures.

Next, we explain the prior works in detail.

### 1.2 Previous Techniques and Their Limitations

We briefly review the works [16] and [9] in order to explain why they have conflicting demands. As they are both based on Barak's non black-box simulation technique, we start by describing this technique.

**Barak's Non-Black-Box Zero Knowledge [1].** Barak's ZK protocol for an NP language $L$ is based on the following idea: a verifier is convinced if one of the two statements is true: (1) the prover knows the verifier's next-message-function, or (2) $x \in L$. By definition a non-black-box simulator knows the code of the next-message-function of the verifier $V^*$ which is a witness for statement 1, while the honest prover has the witness for statement 2. Soundness follows from the fact that no adversarial prover can predict the next-message-function of the verifier. Zero-knowledge can be achieved by employing a witness-indistinguishable (WI) proof for the above statements. The main bottleneck in translating this beautiful idea into a concrete construction is the size of statement 1. Since zero-knowledge demands simulation of arbitrary malicious non-uniform PPT verifiers, there is no *a priori* bound on the size of the verifier's next-message circuit and hence no strict-polynomial bound on the size of the witness for statement 1. Barak and Goldreich [2] in their seminal work show how to construct a WI argument that can *hide* the size of the witness. More precisely, they rely on Universal Arguments (UARG) that can be constructed based on collision-resistant hash-functions (CRHFs) via Probabilistically Checkable Proofs (PCPs) and Merkle hash trees (based on [21]). PCPs allow rewriting of proofs of NP statements in such a way that the verifier needs to check only a few bits to be convinced of the validity of the statement. Merkle hash-trees [23], on the other hand, allow committing to strings of arbitrary length with the additional property that one can selectively open some bits of the string by revealing only a small *fixed* amount of decommitment information. More precisely, a Merkle hash-tree is constructed by arranging the bits of the string on the leaves of a binary tree, and setting each internal node as the hash of its two children: a Merkle tree is a *hash chain*. The commitment to the string corresponds to the commitment to the root of the tree. The decommitment information required to open a single bit of the string is the path from the corresponding leaf in the tree to the root along with their siblings. This path is called *authentication* path. To verify the decommitment, it is sufficient to perform a *consistency check* on the path with the root previously committed. Namely, for each node along the path check if the node corresponds to the hash of the children, till the last node that must correspond to the root. Merkle trees allow for committing strings of super-polynomial length with trees of poly-logarithmic depth. Thus, one can construct a universal argument by putting PCP and Merkle tree together as follows. First, the prover commits to the PCP-proof via a Merkle hash-tree. The verifier responds with a sequence of locations in the proof that it needs to check and the prover opens the bits in the respective locations along with their authentication paths. While this approach allows constructing arguments for statements of arbitrary polynomial size, it is not witness indistinguishable because the authentication paths reveal the size of the proof, and therefore the witness used (this is because the length of the path reveals the depth of the tree). To obtain witness indistinguishability Barak's construction prevents the prover from revealing the actual values on any path. Instead, the prover commits to the paths padded to a fixed length, and then proves that the opening of the commitments corresponds to *consistent* paths

leading to *accepting* PCP answers. A standard ZK is sufficient for this purpose as the size of the statement is strictly polynomial (is fixed as the depth of the Merkle tree). Such ZK proofs, however, need the code of the CRHFs used to build the Merkle-hash tree.

**Black-box Implementation of Barak's Non-black-box Simulation Strategy [16].** Recently, Goyal, Ostrovsky, Scafuro and Visconti in [16] showed how to implement Barak's simulation technique using the hash function in a black-box manner.

They observe that in order to use a hash function in a black-box manner, the prover cannot prove the consistency of the paths by giving a proof, but instead it should reveal the paths and let the verifier recompute the hash values on its own and verify the consistency with the root of the tree. The problem is that to pass the consistency check an honest prover can open paths that are at most as long as the real tree, therefore revealing its size. Instead we need to let the verifier check the path while still keeping the size of the real tree hidden.

To tackle this, they introduce the concept of an *extendable* Merkle tree, where the honest prover will be able to extend any path of the tree *on the fly*, if some conditions are met. Intuitively, this solves the problem of hiding the size of the tree — and therefore the size of the proof— because a prover can show a path for any possible proof length.

To implement this idea they construct the tree in a novel manner, as a sequence of *"LEGO"* nodes, namely nodes that can be connected to the tree on the fly. More concretely, observe that checking the consistency of a path amounts to checking that each node of the path corresponds to the hash of the children. This is a *local* check that involves only 3 nodes: the node that we want to check, say $\mathsf{A}$, and its two children say $\mathsf{leftA}, \mathsf{rightA}$, and the check passes if $\mathsf{A}$ is the hash of $\mathsf{leftA}, \mathsf{rightA}$. The LEGO idea of [16] consists of giving the node the following structure: a node $\mathsf{A}$ now has two fields: $\mathsf{A} = [label, encode]$, where *label* is the hash of the children $\mathsf{leftA}, \mathsf{rightA}$, while *encode* is some suitable encoding of the *label* that allows for black-box proof. Furthermore, *label* is not the hash of the *label* part of $\mathsf{leftA}, \mathsf{rightA}$, but it is the hash of the *encode* part. Thus there is not direct connection between the hash values, and the hash chain is broken. The second ingredient to guarantee binding, is to append to each node a proof of the fact that "*encode* is an encoding of *label*". Note that now the tree so constructed is not an hash chain anymore, the chain is broken at each step. However, it is still binding because there is a proof that connects the hash with its encoding. More importantly note that, if one can cheat in the proof appended to a node, then one can replace/add nodes. Thus, provided that we introduce a *trapdoor* for the honest prover (and the simulator) to cheat in this proof, the tree is *extendable*. Therefore, now we can let the verifier check the hash value by itself, and still be able to hide the depth of the committed tree.

For the honest prover, which *does not* actually compute the PCP proof, the trapdoor is the witness for the theorem "$x \in L$". For the simulator, which is honestly computing the PCP proof and hence the Merkle tree, the trapdoor is the depth of the real tree, i.e., the depth, say $d^*$, at which the leaves of the real

Merkle tree lies. To guarantee binding, the depth $d^*$ is committed at the very beginning. The simulator is allowed to cheat for all PCP queries associated to trees that are not of the committed depth.

The final piece of their construction deals with committing the code of $V^*$ which also can be of an arbitrary polynomial size. The problem is that to verify a PCP proof, the standard PCP verifier needs to read the entire statement (in this case the code of $V^*$) or at least the size of the statement to process the queries. But revealing the size will invalidate ZK again. [16] get around this problem by relying on Probabilistically-Checkable Proof of Proximity (PCPP) [4] instead of PCPs which allow the verifier to verify any proof and arbitrary statement by querying only a few bits of the statement as well as the proof. For convenience of the reader, we provide a very simplified version of [16]'s protocol in Fig. 1.

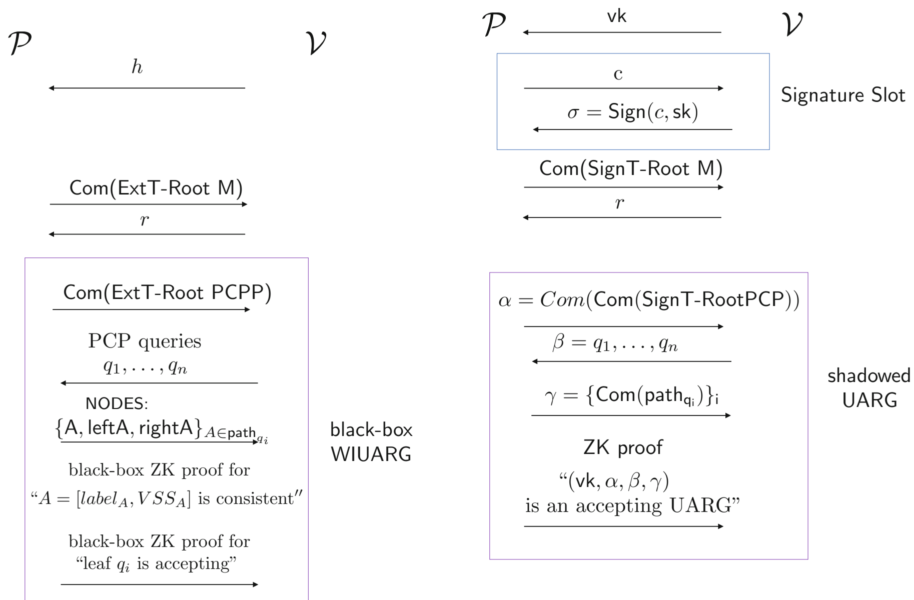


**Fig. 1.** Semi black-box resettably-sound ZK AoK from CRHF [16]

**Fig. 2.** Non-black-box resettably-sound ZK AoK from OWFs [9]

Summing up, some of the key ideas that allow [16] for a black-box use of the hash function are: (1) the prover unfolds the paths of the Merkle tree so that the verifier can directly check the hash consistency, (2) the prover/simulator can arbitrarily extend a path on the fly by computing fake LEGO nodes and cheat in the proof of consistency using their respective trapdoors. The work of [16] retains all the properties of Barak's ZK protocol, namely, is public-coin and constant-round (therefore resettable sound due to [3]), and relies on the underlying CRHF in a black-box manner.

**Non black-box simulation using OWFs [9].** Chung, Pass and Seth [9] showed how to implement the non-black-box simulation technique of Barak using OWFs.

The main idea of CPS is to notice that digital signature schemes — which can be constructed from one-way functions — share many of the desirable properties of CRHFs, and to show how to appropriately instantiate (a variant of) Barak's protocol using signature schemes instead of using CRHFs. More precisely, CPS show that by relying on strong fixed-length signature schemes, one can construct a *signature tree* analogous to the Merkle hash-tree that allows compression of arbitrary length messages into fixed length commitments and additionally satisfies an analogue collision-resistance property. The soundness of such a construction will have to rely on the unforgeability (i.e. collision-resistance) of the underlying signature scheme. Hence it must be the case that is the verifier the one generating the secret key and the signatures for the prover. Towards this, CPS adds a signature slot at the beginning of the protocol. More precisely, first the verifier generates a signature key-pair $\mathsf{vk}, \mathsf{sk}$ and sends only the verification key $\mathsf{vk}$ to the prover. Next, in a "signature slot", the prover sends a commitment $c$ to the verifier, and the verifier returns a valid signature $\sigma$ of $c$ (using $\mathsf{sk}$). The simulator constructs the signature tree by rewinding the (malicious) verifier, and then succeeds in the WIUARG proof as in Barak's protocol. While the simulator can use the signature slot to construct the signature tree, we cannot require the honest prover to construct any tree since it is not allowed to rewind the verifier. To address this, CPS uses a variant of Barak's protocol due to Pass and Rosen [26], which relies on a special-purpose WIUARG, in which the honest prover never needs to perform any hashing. The idea here is that since there exist public-coin UARGs, the prover can first engage in a shadowed UARG where the prover merely commits to its messages and then in a second phase proves using a witness-indistinguishable proof that either $x \in L$ or the messages it committed to constitute a valid UARG. This will allow the honest prover to "do nothing" in the shadowed UARG and use the witness corresponding to $x$ in the second phase. The simulator instead is able to compute a valid signature-tree by rewinding the verifier, and it commits to valid messages in the shadowed UARG.

The resulting protocol is not public-coin, nevertheless [9] shows that it suffices to apply the PRF transformation of BGGL to obtain a protocol that is resettably-sound. We provide a very informal pictorial description of the CPS protocol in Fig. 2. It seems inherent that the CPS protocol needs a shadowed UARG, and hence proving anything regarding this shadowed argument needs to use the underlying OWF in a non-black-box manner.

**Competing requirements of [16] and [9].** In summary, in [16], in order to use the CRHF in a black-box manner, the prover is required to open the paths of the Merkle Tree corresponding to the PCPP queries and let the verifier check their consistency. To preserve size-hiding, the prover needs the ability to arbitrarily extend the paths of the tree by *privately* generating new nodes and this is possible because the prover can compute the hash function on its own. In contrast, in [9] the prover cannot compute nodes on its own, but it needs to get

signatures from the verifier. Therefore the protocol in [9] is designed so that the prover never has to use the signatures.

In this work we show a technique for using the benefits of the signature while relying on the underlying OWF in a black-box manner and we explain this in the next section.

### 1.3 Our Techniques

Our goal is to implement the resettably-sound ZK protocol based on Barak's non-black-box simulation technique using OWFs in a black-box manner. We have illustrated the ideas of [16] to implement Barak's ZK protocol based on extendable Merkle hash-tree that uses a CRHF only as a black-box, and the ideas of [9] that show how to compute a Merkle signature-tree based on (non-black-box use of) OWFs.

The natural first step to reach our goal is then to take the protocol of [16] and implement the extendable Merkle tree with signatures instead of CRHF. As mentioned earlier, this replacement cannot work because the crucial property required to extend the tree is that the prover computes nodes on its own. If we replace CRHF with signatures, then the prover needs to ask signatures from the verifier for every node. This means that any path computed by the prover is already known by the verifier (even the concept of "opening a path" does not seem to make much sense here as the verifier needs to participate in the computation of the path). But instead we need the ability to commit to a tree and then extend it *without* the verifier knowing that we are creating new nodes.

We are able to move forward using the following facts underlying [16]'s protocol. First, although the prover is required to extend paths and prove consistency, it does so by cheating in every proof of consistency of the nodes. Indeed, recall that a node is a pair $(label, encode)$, the consistency between *label* and *encode* is proved via ZK, and the prover cheats in this proof using the witness for "$x \in L$" as a trapdoor.

Under closer inspection we notice that the prover does not need to compute any tree; it just needs to compute the paths corresponding to the PCPP queries on-the-fly when it is asked for it. Indeed, an equivalent version of [16]'s protocol would be the following. The prover commits to the root by just committing to a random string.[1] Then, when it sees the PCPP queries $q_1, \ldots, q_n$ it computes *on-the-fly* the paths for leaves in the corresponding positions, and is able to prove consistency of the paths with the previously committed root by cheating in the proofs. Finally, we point out that the hash function is required only to compute the *label* part of the node, while the *encode* part can be computed by the prover on its own.

Armed with the above observations, we present our idea. As in [9] we place a signature slot at the very beginning of the protocol. This signature slot enables

[1] In [16] the prover actually commits to the hash of two random nodes. In our paper instead we use instance-dependent trapdoor commitments, computed on the instance $x$. The prover, that knows the witness $w$, can just commit to a random string and then equivocate later accordingly.

the simulator to get unbounded number of signatures by rewinding the verifier, and ultimately to construct the extendable Merkle trees. After the signature slot, the prover commits, using an instance-dependent trapdoor commitment scheme, to the roots of the extendable Merkle trees, one tree for the hidden statement of the PCPP (the code of the verifier) and one tree for the PCPP proof. Such trapdoor commitment allows the possessor of the witness to equivocate any commitment. Therefore, the roots committed by the prover – who knows the witness – are not binding. Next, when the prover receives the set of PCPP queries $q_1, \ldots, q_n$ (more specifically, one set of queries for each possible depth of the tree), it computes the paths on-the-fly with the help of the verifier. Namely, for each node along the paths for leaves $q_1, \ldots, q_n$, the prover computes the *encoding* part (which are equivocal commitments), and sends them to the verifier who computes the *label* part by "hashing" the encodings, namely, by computing their signature (in a proper order). Therefore, deviating from [9], we introduce a second signature slot where the prover gets the signatures required to construct the paths dictated by the PCPP queries. Once the labels/ signature for each node have been computed by the verifier, the paths are finally complete. Now the prover can proceed with proving that the paths just computed are *consistent* with the roots previously committed and lead to accepting PCPP answers (such black-box proof follows the ideas of [16]). Interestingly in this game the verifier knows that the prover is cheating and the paths cannot be possibly consistent, but is nevertheless convinced because the only way the prover can cheat in the consistency proof is by using the witness for $x \in L$, which guarantees the validity of $x$.

*Remark 1.* We remark that the prover does not compute any PCPP proof. In the entire protocol it just commits to random values and only in the last step it equivocates the commitments in such a manner that will convince the verifier in the proofs of consistency.

Now, let's look at the simulation strategy. The simulator honestly computes extendable Merkle signature-trees to commit to the machine $V^*$ and to the PCPP, using the first signature slot. Then, when the verifier sends PCPP queries the simulator answers in the following way. For the queries that do not concern the real tree (recall that virtually there are polynomially many possible trees and the PCPP proof can lie in any of those, thus the verifier will provide queries $q_1, \ldots, q_n$ for each possible depth of the tree and expects correct answer only for the queries associated to the depth committed at the beginning) the simulator sends *encode* parts which are commitments of random strings. Later on it will cheat in the proof of consistency for such queries by using its *trapdoor* (as in [16] the trapdoor for the simulator is the size of the real tree). For the queries that hit the real tree, the simulator sends the same commitments that it sent in the first signature slot and that were used to compute the real tree. Indeed, for these nodes the simulator must prove that they are truly consistent, and it can do so by forcing in the view of the verifier the same paths that it already computed for the real tree.

Thus, for the simulation to go through it should be the case that the signatures that the simulator is collecting in the second signature slot, match the ones that it obtained in the first signature slot and that were used to compute the real tree.

Unfortunately, with the protocol outlined above we do not have such guarantee. This is due to two issues. First, the verifier can answer with some probability in the first slot and with another probability in the second slot, therefore skewing the distribution of the output of the simulator. Second, the verifier might not compute the signature deterministically: in this case the signatures obtained in the first slot and used to compute the real tree will not match the signatures obtained in the second slot, where the paths are "re-computed", and thus the simulator cannot prove that the nodes are consistent. We describe the two issue in details and we show how we change the construction to fix each issue.

**Issue 1. $V^*$ aborts in the two signature slots with different probabilities.** We describe this issue with the following example. Consider a malicious verifier that aborts on all messages that start with bit 0 in the first signature slot and aborts on all sets of messages in the second slot if 3/4 of them start with bit 1. The honest prover will succeed with probability close to a 1/2 since the verifier will abort in the first message with probability 1/2 and not abort with high probability in the second slot.[2] The simulator on the other hand can only obtain signatures of commitments that start with bit 1 in the first slot and has to use the same commitments in the second slot. This means that all the commitments sent by the simulator in the second slot will begin with the bit 1 and the verifier will always abort. Hence the simulator can never generate a view. The way out is to come up with a simulation strategy ensuring that the distribution fed in the second slot is indistinguishable to the honest prover's messages.

*Fixing for issue 1.* We first amplify the probability that the verifier gives a signature in the first signature slot by requesting the verifier to provide signatures for $T = O(n^{c+1})$ random commitments instead of just one. Using a Yao-type hardness-amplification, we can argue that if the verifier provides valid signatures with non-negligible probability then we can obtain signatures for at least $1 - \frac{1}{n^c}$ fraction of random tapes for the commitment. Lets call these random tapes $\mathsf{good}$. Now if $k << n^c$ commitments are sent in the second slot, with probability at least $1 - \frac{k}{n^c}$ over random commitments made in the second slot, all of them will be $\mathsf{good}$. This is already promising since the verifier at best will try to detect $\mathsf{good}$ messages in the second slot and with high probability all of them are good. However there is still a non-negligible probability (i.e. $\frac{k}{n^c}$) that the verifier could abort in this simulation strategy. To fix the next issue, we will have the verifier use several keys to sign and we will leverage that to handle this non-negligible fraction of bad messages.

**Issue 2. $V^*$ Does not Compute Signatures Deterministically.** Our protocol requires the verifier to compute the signature deterministically, namely, the randomness used to compute the signatures must be derived from a PRG

[2] We assume here that random commitments are equally likely to have their first bits 0 or 1.

whose seed is sampled along with the parameters $\mathsf{vk}, \mathsf{sk}$ and is part of the secret key. However, in the construction that we outlined before we cannot enforce a malicious verifier from signing deterministically. As mentioned before, if the verifier gives different (correct) signatures for the same node in the first and second slot, the simulator cannot proceed with the proof of consistency. The only way to catch the verifier is to demand from the verifier a proof that the signatures are computed deterministically. Because we need to use the cryptographic primitives in a black-box manner, we cannot solve the problem by attaching a standard proof of consistency.

*Fixing for issue 2.* We force the verifier to be honest using a cut-and-choose mechanism. At high level, we require the verifier to provide signatures for $n$ different keys instead of just one in the signature slots. Together with sending the verification key $\mathsf{vk}_i$, the verifier will also append the commitment to the randomness used in the key generation algorithm (the randomness determines the PRG seed that is used to deterministically sign the messages).

After the first signature slot, the prover asks the verifier to reveal $n/2$ keys (by requiring the verifier to decommit to the randomness used to generate the keys) and checks if, for the revealed keys, the signatures obtained so far were computed honestly. This verification requires the use of OWF only in a black-box manner. The prover proceeds with the protocol using the remaining half of the keys, namely by committing to $n/2$ roots, and obtaining $n/2$ sets of PCPP queries from the verifier. Later, after the second signature slot is completed, the prover will ask to open half of the remaining keys, namely $n/4$ keys, and checks again the consistency of the signatures obtained so far. If all checks pass, the prover is left with paths for $n/4$ trees/PCPP queries for which he has to prove consistency. Due to the cut-and-choose, we know that most of the remaining signatures were honestly generated, but not all of them. Therefore, at this point we will not ask the prover to prove consistency of all remaining $n/4$ executions. Instead, we allow the prover to choose one coordinate among the $n/4$ left, and to prove that the tree in this coordinate is consistent with the paths.

Allowing the prover to choose the tree for which to prove consistency, does not give any advantage compared to the original solution where there was only one tree. On the other hand, having a choice in the coordinate allows the simulator to choose the tree for which it received only consistent signatures and for which it will be able to provide a consistency proof. One can see this coordinate as another *trapdoor*. Namely, in the previous construction, the simulator commits to the depth of the real tree, so that in the consistency proof it can cheat in answering all queries that do not hit the committed value. We follow exactly the same concept by adding the commitment of the coordinate. The simulator commits to the coordinate for which it wants to provide the proof, and it is allowed to cheat in all the remaining proofs. Finally, because we are in the resettable setting, we require the prover to commit in advance to the coordinates that he wants the verifier to open. If this was not the case then the prover can rewind the verifier and get all the secret keys.

It only remains to argue that there is a strategy so that the simulator can always find one good key. Recall that the simulator rewinds the verifier several times in the first signature slot to obtain the signature on any message. The adversary has to sign deterministically in most coordinates because of the cut-and-choose mechanism. However, it can cheat in a few of them and in different rewindings it can choose to cheat in different keys. To ensure that the signatures obtained are the ones that are deterministically signed, we will make the simulator to obtain $n$ signatures on a commitment and take that signature that occurs more than half of the times. However, there could be messages for which there will be no majority among the $n$ signatures or worse the wrong signature in majority. This issue is quite subtle and combining the Yao-amplification and the cut-and-choose mechanism we argue that for all but small fraction of the messages, the simulator will obtain the deterministically-signed signature for most of the unopened keys. As with the first issue, we are still left with a small fraction of messages for which the simulator could receive a bad signature.

**Handling Bad Messages and Bad Signatures.** The fix for Issue 1 results in the simulator using a small fraction ($\approx \frac{1}{poly(n)}$) of commitments in the actual Merkle Tree whose signatures are not good messages w.r.t the first Signature Slot. If such a message is fed in the second Signature Slot, the Verifier can detect it. The fix for Issue 2 results in the simulator using a small fraction of commitment with bad signatures (i.e., not deterministically signed). If a message for which a bad signature was obtained is fed in the second Signature Slot then the signature obtained in the second slot will be different from the bad signature that the simulator obtained. However, we need to guarantee that the simulator will be successful in placing the commitments used in first signature slot to compute the Merkle tree, into the second signature slot, for at least one index among the $n/4$ trees remaining from the cut-and-choose. It will "test the waters" first: More precisely, before feeding the actual commitments in the second Signature Slot, for every unopened key, it will first generate random commitments to be sent in the second Signature Slot for that key and check if the random commitments are good messages w.r.t the first signature slot. More precisely, it will check if the first slot yields a signature for these commitments. A key is considered good if all these random commitments turn out to be good. For good keys, the simulator will swap the commitments with the actual commitments used to generate the Merkle Trees. It can be shown that the distribution induced by this swap is not skewed because a set of random good messages are swapped with the real commitments which are also random good commitments, since they yielded signatures in the first Signature Slot. This will help us handle bad messages as long as we can show there will be good keys. Arguing this turns out to be subtle and our proof will show that there will be only few bad keys. An analogous argument can be made to show there will be only few bad keys for which there is some commitment among the ones to be sent in the second Signature Slot by the simulator with a bad signature. If we start with sufficiently many ($O(n)$) keys then we will be able to show that there will be at least one good key that survives from bad messages and bad signatures.

## 2 Definitions

In this section we provide the definitions of some of the tools that we use in our construction. We refer the reader to the full version [25] for more details and for the definition of more standard tools, like instance-dependent equivocal commitment, that we omit in this section. We assume familiarity with interactive arguments and argument of knowledge.

**Definition 1 (Zero-knowledge [13]).** *An interactive protocol $(P, V)$ for a language $L$ is* zero-knowledge *if for every PPT adversarial verifier $V^*$ and auxiliary input $z \in \{0,1\}^*$, there exists a PPT simulator $S$ such that the following ensembles are computationally indistinguishable over $x \in L$:*

$$\{\mathsf{View}_{V^*}\langle P, V^*(z)\rangle(x)\}_{x \in L, z \in \{0,1\}^*} \approx \{S(x,z)\}_{x \in L, z \in \{0,1\}^*}$$

**Definition 2 (Resettably-sound Arguments [3]).** *A resetting attack of a cheating prover $P^*$ on a resettable verifier $V$ is defined by the following two-step random process, indexed by a security parameter $n$.*

1. *Uniformly select and fix $t = poly(n)$ random-tapes, denoted $r_1, \ldots, r_t$, for $V$, resulting in deterministic strategies $V^{(j)}(x) = V_{x,r_j}$ defined by $V_{x,r_j}(\alpha) = V(x, r_j, \alpha)$,[3] where $x \in \{0,1\}^n$ and $j \in [t]$. Each $V^{(j)}(x)$ is called an incarnation of $V$ .*
2. *On input $1^n$, machine $P^*$ is allowed to initiate $poly(n)$-many interactions with the $V^{(j)}(x)$'s. The activity of $P^*$ proceeds in rounds. In each round $P^*$ chooses $x \in \{0,1\}^n$ and $j \in [t]$, thus defining $V^{(j)}(x)$, and conducts a complete session with it.*

*Let $(P, V)$ be an interactive argument for a language $L$. We say that $(P, V)$ is a* resettably-sound argument *for $L$ if the following condition holds:*

– Resettable-soundness*: For every polynomial-size resetting attack, the probability that in some session the corresponding $V^{(j)}(x)$ has accepted and $x \notin L$ is negligible.*

Similarly to [9,8] we consider the following weaker notion of resettable soundness, where the statement to be proven is fixed, and the verifier uses a single random tape (that is, the prover cannot start many independent instances of the verifier).

**Definition 3 (Fixed-input Resettably-sound Arguments [27]).** *An interactive argument $(P, V)$ for a* **NP** *language $L$ with witness relation $\mathcal{R}_L$ is* fixed-input resettably-sound *if it satisfies the following property: For all non-uniform polynomial-time adversarial prover $P^*$, there exists a negligible function $\mu(\cdot)$ such that for every all $x \notin L$,*

$$\Pr[\mathsf{ran} \leftarrow \{0,1\}^\infty; (P^{*V_{\mathsf{ran}}(x,\mathsf{pp})}, V_{\mathsf{ran}})(x) = 1] \leq \mu(|x|)$$

[3] Here, $V(x, r, \alpha)$ denotes the message sent by the strategy $V$ on common input $x$, random-tape $r$, after seeing the message-sequence $\alpha$.

This is sufficient because it was shown in [9] that any zero-knowledge *argument of knowledge* satisfying the weaker notion can be transformed into one that satisfies the stronger one, while preserving zero-knowledge (or any other secrecy property against malicious verifiers.

*Claim.* Let $(P, V)$ be a fixed-input resettably sound zero-knowledge (resp. witness indistinguishable) argument of knowledge for a language $L \in \mathbf{NP}$ . Then there exists a protocol $(P', V')$ that is a (full-fledged) resettably-sound zero-knowledge (resp. witness indistinguishable) argument of knowledge for $L$.

**Strong Deterministic Signature.** In this section we define strong, fixed-length, deterministic secure signature schemes that we rely on in our construction. Recall that in a strong signature scheme, no polynomial-time attacker having oracle access to a signing oracle can produce a valid message-signature pair, unless it has received this pair from the signing oracle. The signature scheme being fixed-length means that signatures of arbitrary (polynomial-length) messages are of some fixed polynomial length. Deterministic signatures do not use fresh randomness in the signing process once the signing key has been chosen. In particular, once a signing key has been chosen, a message $m$ will always be signed the same way.

**Definition 4 (Strong Signatures).** *A* strong, length-$\ell$, signature scheme $\mathsf{SIG}$ *is a triple* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *of* $\mathsf{PPT}$ *algorithms, such that*

1. *for all* $n \in \mathcal{N}, m \in \{0,1\}^*$,

$$\Pr[(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^n), \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}](m); \mathsf{Ver}_{\mathsf{vk}}(m, \sigma) = 1 \wedge |\sigma| = \ell(n)] = 1$$

2. *for every non-uniform* $\mathsf{PPT}$ *adversary* $A$*, there exists a negligible function* $\mu(\cdot)$ *such that for all* $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^n)$ *it holds:*

$$\Pr[(m, \sigma) \leftarrow A^{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}(1^n); \mathsf{Ver}_{\mathsf{vk}}(m, \sigma) = 1 \wedge (m, \sigma) \notin L] \leq \mu(n),$$

*where* $L$ *denotes the list of query-answer pairs of* $A$*'s queries to its oracle.*

Strong, length-$\ell$, **deterministic** signature schemes with $\ell(n) = n$ are known based on the existence of OWFs; see [24,30,12] for further details. In the rest of this paper, whenever we refer to signature schemes, we always means strong, length-$n$ signature schemes.

Let us first note that signatures satisfy a "collision-resistance" property.

*Claim.* Let $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ be a strong (length-$n$) signature scheme. Then, for all non-uniform PPT adversaries $A$, there exists a negligible function $\mu(\cdot)$ such that for every $n \in \mathcal{N}$, for all $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^n)$ it holds:

$$Pr[(m_1, m_2, \sigma) \leftarrow A^{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}(1^n, \mathsf{vk}); \mathsf{Ver}_{\mathsf{vk}}(m_1, \sigma) = \mathsf{Ver}_{\mathsf{vk}}(m_2, \sigma) = 1] \leq \mu(n)$$

**Verifiable Secret Sharing (VSS).** A verifiable secret sharing scheme (VSS for short) [7] is a two-stage protocol run among $n+1$ players. In the first stage, called $\texttt{Share}(s)$, a special player, referred to as dealer, distributes a string $s$ among the $n$ players so that any $t$ players (where $t = n/c$ for some constant $c > 3$) colluding cannot reconstruct the secret. The output of the $\texttt{Share}$ phase is a set of VSS views $\mathsf{S}_1, \ldots, \mathsf{S}_n$ that we call VSS shares. In the second stage, called $\texttt{Recon}(\mathsf{S}_1, \ldots, \mathsf{S}_n)$, any $(n-t)$ players can reconstruct the secret $s$ by exchanging their VSS shares. The scheme guarantees that if at most $t$ players are corrupted the $\texttt{Share}$ stage is hiding, moreover a dishonest dealer is caught at the end of the $\texttt{Share}$ phase through an accusation mechanism that disqualifies the dealer (this property is called $t$-privacy). A VSS scheme can tolerate errors on malicious dealer and players distributing inconsistent or incorrect shares, indeed the critical property is that even in case the dealer is dishonest but has not been disqualified, still the second stage always reconstructs the same string among the honest players.

**MPC-in-the-head.** MPC-in-the-head is a breakthrough technique introduced by Ishai at al. in [20] to construct a black-box zero-knowledge protocol. Let $\mathcal{F}_{ZK}$ be the zero-knowledge functionality for an NP language L, that takes as public input $x$ and one share from each player $P_i$, and outputs 1 iff the secret reconstructed from the shares is a valid witness. Let $\mathsf{MPCZK}$ be a perfect $(t, n)$-secure MPC protocol implementing $\mathcal{F}_{ZK}$.

Very roughly, the "MPC-in-the-head" idea is the following. The prover runs *in his head* an execution of a $(t, n)$-secure $\mathsf{MPCZK}$ protocol among $n$ imaginary players, each one participating in the protocol with a share of the witness. Then it commits to the view of each player separately. The verifier obtains $t$ randomly chosen views, and checks that such views are consistent with an honest execution of the protocol and accepts if the output of every player is 1. Clearly $P^*$ decides the randomness and the input of each player so it can cheat at any point and make players output 1. However, the crucial observation is that in order to do so, it must be the case that a constant fraction of the views committed are not consistent (this property is called $t$-robustness). Thus by selecting the $t$ views at random, $V$ will catch inconsistent views whp.

One can extend this technique further (as in [15]), to prove a general predicate $\phi$ about arbitrary values. Namely, one can consider the functionality $\mathcal{F}_\phi$ in which every player $i$ participates with an input that is a view of a VSS player $\mathsf{S}_i$. $\mathcal{F}_\phi$ collects all such views, and outputs 1 if and only if $\phi(\texttt{Recon}(\mathsf{S}_1, \ldots \mathsf{S}_n)) = 1$. This idea is crucially used in [16].

**Probabilistically Checkable Proofs.** Informally, a PCP [2] system for a language $L$ consists of a proof $\pi$ written in a redundant form for a statement "$x \in L$", and a PPT verifier, which is able to decide the truthfulness of the statement by reading only few bits of the proof.

A PCP verifier $V$ can be decomposed into a pair of algorithms: the query algorithm $\mathsf{Q}_{\mathsf{pcp}}$ and the decision algorithm $\mathsf{D}_{\mathsf{pcp}}$. $\mathsf{Q}_{\mathsf{pcp}}$ on input $x$ and random tape $r$, outputs positions $q_1 = \mathsf{Q}_{\mathsf{pcp}}(x, r, 1), q_2 = \mathsf{Q}_{\mathsf{pcp}}(x, r, 2), \ldots, q_n = \mathsf{Q}_{\mathsf{pcp}}(x, r, p(|x|))$, for some polynomial $p$, and the prover answers with $b_i = \pi[q_i]$. $V$ accepts if

$\mathsf{D}_{\mathsf{pcp}}(x, r, b_1, \ldots, b_{p(|x|)})$ outputs 1. For later, it is useful to see algorithm $\mathsf{D}_{\mathsf{pcp}}$ as a predicate defined over a string $\pi$ which is tested on few positions.

**Probabilistically Checkable Proofs of Proximity.** The standard PCP verifier decides whether to accept the statement $x \in L$ by probing few bits of the proof $\pi$ and reading the entire statement $x$. A "PCP of proximity" (PCPP) [4] is a relaxation of PCP in which the verifier is able to make a decision without even reading the entire statement, but only few bits of it. More specifically, in a PCPP the theorem is divided in two parts $(a, y)$. A public string $a$, which is read entirely by the verifier, a private string $y$, for which the verifier has only *oracle* access. Consequently, PCPP is defined for pair languages $L \subset \{0,1\}^* \times \{0,1\}^*$. For every $a \in \{0,1\}^*$, we denote $L_a = \{y \in \{0,1\}^* : (a, y) \in L\}$. The PCP Verifier can be seen as a pair of algorithms $(\mathsf{Q}_{\mathsf{pcpx}}, \mathsf{D}_{\mathsf{pcpx}})$, where $\mathsf{Q}_{\mathsf{pcpx}}(a, r, i)$ outputs a pair of positions $(q_i, p_i)$: $q_i$ denotes a position in the theorem $y$, $p_i$ denotes a position in the proof $\pi$. $\mathsf{D}_{\mathsf{pcpx}}$ decides whether to accept $(a, y)$ by looking at the public theorem $a$, and at positions $y[q_i]$, $\pi[p_i]$. For later, it is useful to see algorithm $\mathsf{D}_{\mathsf{pcpx}}$ as a predicate defined over two strings $y, \pi$, testing few positions of each string.

**Definition 1 (PCPP verifier for a pair language).** *For functions $s, \delta : \mathcal{N} \to [0, 1]$, a verifier $V$ is a probabilistically checkable proof of proximity (PCPP) system for a pair language $L$ with proximity parameter $\delta$ and soundness error $s$, if the following two conditions hold for every pair of strings $(a, y)$:*

- *Completeness: If $(a, z) \in L$ then there exists $\pi$ such that $V(a)$ accepts oracle $y \circ \pi$ with probability 1. Formally:*

$$\exists \pi, \Pr_{(\mathsf{Q},\mathsf{D}) \leftarrow V(a)}[\mathsf{D}((y \circ \pi)|_{\mathsf{Q}}) = 1] = 1.$$

- *Soundness: If $y$ is $\delta(|a|)$-far from $L(a)$, then for every $\pi$, the verifier $V(a)$ accepts oracle $y \circ \pi$ with probability strictly less than $s(|a|)$. Formally:*

$$\forall \pi, \Pr_{(\mathsf{Q},\mathsf{D}) \leftarrow V(a)}[\mathsf{D}((y \circ \pi)|_{\mathsf{Q}})] = 1 < s(|a|).$$

Note that the query complexity of the verifier depends only on the public input $a$ [10].

## 3 Protocol

*Overview.* The protocol starts with a signature slot, where the prover sends $T$ commitments, and the verifier signs all of them. Then, as per Barak's construction, the prover sends a message $z$, which is the commitment to a machine $M$, the verifier sends a random string $r$, and finally the prover sends a commitment to a PCP of Proximity proof for the theorem: "the machine $M$ committed in $z$ is such that $M(z) = r$ in less then $n^{\log n}$ steps". $M$ is the hidden theorem for the PCP of Proximity and the verifier has only oracle access to it. The above commitments are commitment to the roots of (extendable) Merkle signature-trees

(that we will describe in details later). Next, the verifier sends the PCPP queries. As the verifier does not know the length of the PCPP proof and thus the depth of the Merkle tree, it will send a set of PCPP queries for each possible depth. Each PCPP query is a pair of indices, one index for the hidden theorem $M$ and one for the PCPP proof. The verifier expects to see a path for each index.

At this point the prover needs to compute such paths. As we mentioned in the introduction, in a signature tree a path cannot be computed by the prover only: each nodes consists of two parts, the signature of the children, called *label*, and the encoding of the label, that we called *encode*. Thus, the prover continues as follows. For each path that must be provided for a query, he computes on-the-fly the *encode* parts of the nodes belonging to such path. It then sends all these "half" nodes to the verifier. The verifier computes the *label* parts for each node by signing the *encode* part and send them back to the prover. This is the second signature slot. Once all paths are completed, the prover starts the proof stage. Using a ZK protocol he proves that: (1) the paths are consistent with the root committed before, (2) the leaves of the paths open to accepting PCPP answers, in a black-box manner. How does the prover pass the proof stage? The prover computes all commitments using instance-dependent equivocal commitments and later cheats in the opening. How does the simulator pass the proof stage? The simulator computes consistent trees for the machine $V^*$ and the PCPP proof by rewinding the verifier in the first signature slot, and committing to their depth at the beginning. On top of this outlined construction we use cut-and-choose to force the verifier to compute the signatures deterministically, so that the simulator can use the tree computed in the first signature slot. This concludes the high-level description of the protocol.

Now we need to show concretely how to compute the proofs of consistency using the signature and the commitment scheme only as *black-box*. This requires to go into the details of the (extendable) Merkle signature-tree and the mechanism for size hiding introduced in [16]. We provide such details in the next section.

### 3.1 Ingredients of the Construction

We present the ingredients of our construction in this section. Some of the ideas are adapted from [16].

**String Representation.** To allow black-box proofs for a committed string, the first ingredient is to represent the string with a convenient encoding that enables to give black-box proofs on top. For this purpose, following [16,15] we use a $(t, n)$-secure VSS scheme, defined in Sec. 2. To commit to any string $s$, the prover first runs, in his head, an execution of a *perfectly* $(t, n)$-secure VSS among $n + 1$ imaginary players where the dealer is sharing $s$, obtaining $n$ views: $\mathsf{S}[1], \ldots, \mathsf{S}[n]$. Then it commits to each share separately using a statistically binding commitment.

**Black-box Proof of a Predicate.** With the VSS representation of strings, now the prover can commit to the string and prove any predicate about the committed string, using MPC-in-the-head as follow. Let $[\mathsf{S}[1], \ldots, \mathsf{S}[n]]$ be the VSS shares

that reconstruct to a string $s$ and let $\phi$ be a predicate. The prover wants to prove that $\phi(s)$ is true without revealing $s$. Define $\mathcal{F}_\phi$ as the $n$-party functionality that takes in input one VSS share $\mathsf{S}[p]$ from each player $p$, and outputs $\phi(\mathtt{Recon}(\mathsf{S}[1], \ldots, \mathsf{S}[n]))$. To prove $\phi(s)$, the prover runs a $(t, n)$-perfectly secure MPC-in-the-head among $n$ players for the functionality $\mathcal{F}_\phi$ . Each player participates to the protocol with input a VSS share of the string $s$. Then the prover commits to each view of the MPC-in-the-head so computed. The verifier checks the proof by observing $t$ randomly chosen views of *both* the VSS and the MPC protocol, and checking that such views are consistent with an honest execution of the VSS and MPC protocols. Zero-knowledge follows from the $t$-privacy and soundness follows from the $t$-robustness of the MPC/VSS protocols, where $t$-robustness roughly means that, provided that the predicate to be proved is false and that the prover does not know in advance which views will be opened, corrupting only $t$ players is not sufficient to convince the verifier with consistent views. On the other hand, by corrupting more than $t$ players, the prover is caught whp.

**(Extendable) Merkle Signature-Tree.** As we discussed in the introduction, a node in an extendable Merkle tree is a pair $[label, encode]$. In our signature Merkle tree, the field *label* is a vector of signatures (computed by the verifier), and the field *encode* is a vector of commitments of VSS shares of *label*. Specifically, let $\gamma$ be any node, let $\gamma 0 = [label^{\gamma 0}, \{\mathsf{Com}(\mathsf{S}^{\gamma 0}[i])\}_{i \in n}]$ be its left child, and $\gamma 1 = [label^{\gamma 1}, \{\mathsf{Com}(\mathsf{S}^{\gamma 1}[i])\}_{i \in n}]$ be its right child. Node $\gamma$ is computed as follows. The label part is $label^{\gamma} = \{\mathsf{Sign}_{\mathsf{sk}}(\mathsf{Com}(\mathsf{S}^{\gamma b}[i]))\}_{b \in \{0,1\}, i \in n}$. The *encode* part is computed in two steps: First, compute shares $\mathsf{S}^{\gamma}_1, \ldots, \mathsf{S}^{\gamma}_n \leftarrow \mathtt{Share}(label)$; next commit to each share separately. At **leaf level**, the $label^{\gamma} = s[\gamma]$, namely the $\gamma$-th bit that we want to commit.

**Hiding the Size of the Tree.** The size of the string committed, and hence the depth of the corresponding tree, is not known to the verifier and it must remain hidden. Specifically, the verifier should not know the size of the machine $M$ and of the PCPP proof. Hence, the verifier will send a set of PCPP queries for each possible depth of the tree for the PCPP. Namely, for each possible depth $j \in [\log^2 d]$, $V$ sends $\{q_{i,j}, p_{i,j}\}_{i \in k}$[4], where $k$ is the soundness parameter.

Note that the prover (actually, the simulator) commits to one tree and is therefore able to correctly answer only the queries lying on the depth of the committed tree, and we want this to be transparent to the verifier. This is done by adding the commitment to the depth of the real tree at the beginning, and then proving for each query that either the query is correctly answered or the query refers to a depth that is different from the one committed. The commitment of the depth needs to be in the same format of the *encode* part of the nodes (i.e., it will be a commitment of VSS shares) because it will be used in the black-box proofs of consistency.

[4] We assume that the length PCPP proof is a power of 2. Also, for the sake of simplifying the notation we use the same index $j$ for the queries to the machine $q_{i,j}$ and the proof $p_{i,j}$, even though the machine $M$ and the corresponding PCPP proof $\pi$ might lie on different depths.

**Black-box Proofs about the Leaves of the Tree.** As it should be clear by now, the proof consists in a sequence of paths, one for each PCPP query, and a sequence of proofs (one for every node and one for every possible depth) claiming that (1) the paths are consistent with the one committed root; (2) the paths open to accepting PCPP answers. This is done as follows. Attached to each path, there is a proof of consistency. This proof serves to convince the verifier that the path is consistent with the previously committed root. Attached to each set of paths (there is a set of paths for each depth $j = 1, \ldots, \log^2 n$), there is a proof of acceptance. This proof serves to convince the verifier that those paths open to bits of PCPP proof/hidden theorem $M$ that are accepting.

Both the prover and the simulator will cheat in this proof, but in different ways. The prover cheats in all proofs by equivocating the commitments (using the witness as trapdoor). The simulator cheats in all proofs concerning paths that do not match the depth of the real tree, by using the commitments of the depth as a trapdoor. Namely, the simulator will prove that either the path is consistent/accepting, or the depth of such path does not match the depth committed (note that there will exist one path that will match the committed depth). For the paths of the real tree, the simulator will honestly compute the proof. We now describe each step of the proof in more details.

*Proof that a path is consistent.* Let $p_{i,j}$ be a PCPP query for a tree of depth $j$. Associated to this query there is a path. Proving consistency of a path for $p_{i,j}$ amounts to prove consistency of each node along the path. For each node $\gamma$ along the path for $p_{i,j}$, there the $\gamma$ is $\left[label^{\gamma}, \{\mathsf{Com}(\mathsf{S}^{\gamma}[p])\}_{p \in n}\right]$, the prover proves that $\mathtt{Recon}(\mathsf{S}^{\gamma}[1], \ldots, \mathsf{S}^{\gamma}[n]) = label^{\gamma}$.
This is done via an MPC-in-the-head protocol, for a functionality $\mathcal{F}_{\mathsf{innode}}$ that takes in input the share $\mathsf{S}^{\gamma}[p]$ of the label, the share $\mathsf{S}_{\mathsf{depth}}[p]$ of the committed depth, and the string $label^{\gamma}$ and outputs 1 to all players if either $\mathtt{Recon}(\mathsf{S}^{\gamma}[1], \ldots, \mathsf{S}^{\gamma}[n]) = label^{\gamma}$ or $\mathtt{Recon}(\mathsf{S}_{\mathsf{depth}}[1], \ldots, \mathsf{S}_{\mathsf{depth}}[n]) \neq j$. Where $\{\mathsf{S}_{\mathsf{depth}}[1], \ldots, \mathsf{S}_{\mathsf{depth}}[n]\} \leftarrow \mathtt{Share}(\mathsf{depth})$ are the share of the depth of the real tree and were committed at the beginning. The actual proof consists in the commitment of the views of the MPC players. The verifier verifies the proof by opening $t$ views checking their consistency.

*Proof that a set of paths is accepting.* For each level $j$, the verifier sends queries $\{p_{i,j}, q_{i,j}\}_{i \in [k]}$ and the prover opens a path for each query. To prove that these queries are accepting, the prover computes a proof that involves the leaves of the paths of depth $j$. The prover runs an MPC-in-the-head for a functionality, that we call $\mathcal{F}_{\mathsf{VerPCPP}}$, that will check that the values in those positions will be accepted by a PCPP verifier. $\mathcal{F}_{\mathsf{VerPCPP}}$ takes in input shares: $\mathsf{S}^{p_{i,j}}[p]$, $\mathsf{S}^{q_{i,j}}[p]$, $\mathsf{S}_{\mathsf{depth}}[p]$ (with $p = 1, \ldots, n$) and the public theorem; it then reconstructs the bits of the PCPP proof $\pi_{i,j} = \mathtt{Recon}(\mathsf{S}^{p_{i,j}}[1], \ldots, \mathsf{S}^{p_{i,j}}[n])$ and of the hidden theorem $m_{i,j} = \mathtt{Recon}(\mathsf{S}^{q_{i,j}}[1], \ldots, \mathsf{S}^{p_{i,j}}[n])$. It finally outputs 1 to all players iff: either the PCPP verifier accepts the reconstructed bits, i.e., $\mathsf{D}_{\mathsf{pcpx}}(m_{i,j}, \pi_{i,j}, q_{i,j}, p_{j,i}) = 1$, or if $\mathtt{Recon}(\mathsf{S}_{\mathsf{depth}}[1], \ldots, \mathsf{S}_{\mathsf{depth}}[n]) \neq j$. The actual proof consists of the commitment of the views of the MPC players for $\mathcal{F}_{\mathsf{VerPCPP}}$. The verifier verifies the proof by checking the consistency of $t$ views.

*Verification of the proof.* The verifier receives the commitments of all such views and ask the prover to open $t$ of them. The prover will then decommits $t$ views for *all* MPC/VSS protocol committed before as follow: first, it computes accepting MPC views by running the simulator granted by the $t$-security of the MPC-in-the-head protocol, then it open to such views by equivocating the corresponding commitments.

**The Cut-and-choose.** The mechanism described above is repeated $n$ times: the verifier provides $n$ signature keys, and the prover will compute $n$ trees. During the protocol $3/4n$ of the secret keys will be revealed, so for those indexes the prover will not proceed to the proof phase. In fact, the prover will prove consistency of only one tree among the $1/4n$ trees left (but of course, we want the verifier to be oblivious about the tree that the prover is using). Thus, the last ingredient of our construction is to ask the prover to commit to an index $J$ among the remaining $1/4n$ indexes. This commitment is again done via $VSS$ of $J$ and then committing to the view[5]. As expected, such VSS will be used in the computation of $\mathcal{F}_{\mathsf{innode}}$ and $\mathcal{F}_{\mathsf{VerPCPP}}$. The functionalities now will first check if the nodes are part of the $J$-th tree. If not, it means that the tree is not the one that must be checked, in such a case the functionality outputs 1 to all players regardless of whether any condition is satisfied.

### 3.2 The Construction

We now put everything together and provide the description of the final protocol. Some details are omitted for simplicity, a full specification of our protocol can be found in the full version [25]. We remark that in any step of the protocol the randomness used by the verifier to compute its messages is derived by the output of a PRF computed on the entire transcript computed so far. *Common Input:* An instance $x$ of a language $L \in \mathbf{NP}$ with witness relation $\mathbf{R}_L$. *Auxiliary input to $P$:* A witness $w$ such that $(x, w) \in \mathbf{R}_L$.

**Cut-and-choose 1**

- $P_0$: Randomly pick two disjoint subsets of $\{1, \ldots, n\}$ that we denote by $J_1, J_2$, with $|J_1| = n/2$ and $|J_2| = n/4$. Commit to $J_1, J_2$ using the equivocal commitment scheme.
- $V_0$: Run $(\mathsf{sk}_\kappa, \mathsf{vk}_\kappa) \leftarrow \mathsf{Gen}(1^n, r_\kappa)$ for $\kappa = 1, \ldots, n$. Send $\mathsf{vk}_\kappa, \mathsf{Com}(r_\kappa)$ for $\kappa = 1, \ldots, n$ to $P$.
- **Signature Slot 1**. $P_1$: Send $\mathsf{T} = O(n^c)$ commitments using the equivocal commitment scheme to $0^v$ (for some constant $c$ and for $v$ being as the size of the *encode* part). $V_1$: Signs each commitment.
  **Check Signature Slot 1**. $P$ opens set $J_1$. $V$ send $\mathsf{sk}_\kappa$ and decommitment to $r_\kappa$, for $\kappa \in J_1$. $P$ checks that all signatures verified under key $\mathsf{vk}_\kappa$ are consistent with $\mathsf{sk}_\kappa, r_\kappa$. If not, abort.

[5] Attached to the VSS there will be also a proof that proves that $J \in \{1, \ldots, n\}/\{J_1 \cup J_2\}$.

**Commitment to the Machine**

- $P_2$: Send equivocal commitment to the *encode* part of the root of the (extendable) Merkle signature-tree for $M$, and equivocal commitment to the depth of the tree, this is done for each $\kappa \in \{1, \ldots, n\}/J_1$.
- $V_2$: Send a random string $r \in \{0,1\}^n$. Let $(r, \mathtt{t})$ be the public theorem for the PCPP for the language: $\mathcal{L}_{\mathcal{P}} = \{(a = (r, \mathtt{t}), (Y)),\ \exists M \in \{0,1\}^*$ s.t. $Y \leftarrow \mathsf{ECC}(M)$, $M(z) \rightarrow r$ within $\mathtt{t}$ steps$\}$ (where $\mathsf{ECC}(\cdot)$ is a binary error correcting code tolerating a constant fraction $\delta > 0$ of errors, and $\delta$ is the proximity factor of PCPP).

**Commitment to the PCPP proof**

- $P_3$: Send equivocal commitment to the *encode* part of the root of the (extendable) Merkle signature-tree for the PCPP proof and the commitment to the depth of such tree. This is done for each $\kappa \in \{1, \ldots, n\}/J_1$.
- $V_3$: Send the random tapes for the PCPP queries. $V$ and $P$ obtain queries $(q_{i,j}, p_{i,j})$ for $i \in [k]$ and with $j = 1, \ldots, \log^2 n$.
- $P_4$: Send paths for $p_{i,j}, q_{i,j}$. Namely, send the *encode* part for each node along the paths $p_{i,j}, q_{i,j}$. This is done for each tree $\kappa \in \{1, \ldots, n\}/J_1$, previously committed.

**Signature Slot 2** $V_4$: Sign the *encode* parts received from $P$.

**Cut-and-choose 2.** $P$ opens the set $J_2$. $V$ sends $\mathsf{sk}_\kappa$ and decommitment to $r_\kappa$, for $\kappa \in J_2$. $P$ checks that all signatures verifier under key $\mathsf{vk}_\kappa$ are consistent with $\mathsf{sk}_\kappa, r_\kappa$. If not, abort.

**Proof**

- $P_5$: Commit to a random index $J \in \{1, \ldots, n\}/\{J_1 \cup J_2\}$.

  Then compute the proof of consistency of each path, and the proof of acceptance for each set of queries (as explained earlier), for each of the remaining trees. The proofs (i.e., the views of the MPC-in-the-head) are committed using an *extractable* equivocal commitment scheme.
- $V_5$: Select $t$ players to check: Send indexes $p_1, \ldots, p_t$.
- $P_6$: Compute the $t$ VSS shares and the $t$ views of the MPC protocols that will make the verifier accept the proof. This is done by running the simulator guaranteed by the perfect $t$-security of the MPC protocols for the proofs. $P$ then equivocates the previously committed views so that they open to this freshly computed views.
- $V_6$: Accept if all the opened views are consistent and output 1.

## 4 Security Proof

In this section we sketch the proof of the following theorem (for the formal proof the reader is referred to the full version of this work [25]).

**Theorem 2.** *There exists a (semi) black-box construction of a resettably-sound zero-knowledge argument of knowledge based on one-way functions.*

**Resettable Soundness.** We prove *fixed-input* resettable-soundness of the protocol without loss of generality. Assume for contradiction, there exists a $\mathsf{PPT}$ adversary $P^*$, sequences $\{x_n\}_{n\in\mathcal{N}} \subseteq \{0,1\}^*/L$, $\{z_n\}_{n\in\mathcal{N}} \subseteq \{0,1\}^*$ and polynomial $p(\cdot)$ such that for infinitely many $n$, it holds that $P^*$ convinces $V$ on common input $(1^n, x_n)$ and private input $z_n$ with probability at least $\frac{1}{p(n)}$. Fix an $n$, for which this happens.

First, we consider a hybrid experiment HYB, where we run the adversary $P^*$ on input $(x_n, z_n)$ by supplying the messages of the honest verifier with a small modification. For all the randomness used by the verifier in the protocol via a PRF applied on the transcript, we instead supply truly random strings. In particular, the signature keys generated in the first message, the challenge string in message $V_2$, the random strings in $V_3$ and random $t$ indices in $V_5$ are sampled truly randomly. By the pseudo-randomness of the PRF, we can conclude that $P^*$ convinces the emulated verifier in this hybrid with probability at least $\frac{1}{p(n)} - \nu(n) > \frac{1}{2p(n)}$ for some negligible function $\nu(\cdot)$.

The high-level idea is that using $P^*$, we construct an oracle adversary $A$ that violates the collision-resistance property of the underlying signature scheme. In more details, $A$ is an oracle-aided $\mathsf{PPT}$ machine that on input $\mathsf{vk}, n$ and oracle access to a signing oracle $\mathsf{SIG}_{\mathsf{sk}}(\cdot)$ proceeds as follows: It internally incorporates the code of $P^*$ and begins emulating the hybrid experiment HYB by providing the verifier messages. Recall that $P^*$ is a resetting prover and can open arbitrary number of sessions by rewinding the verifier. $A$ selects a random session $i$.

- For all the unselected sessions, $A$ simply emulates the honest verifier.
- For the selected session, $A$ proceeds as follows.
  (1) In the first message of the protocol, $A^{\mathsf{SIG}_{\mathsf{sk}}(\cdot)}$ chooses a random index $f \in [n]$ and places the $\mathsf{vk}$ in that coordinate. More precisely, it sends $(\mathsf{vk}_f, = \mathsf{vk}, c)$ where $c$ is a commitment to the 0 string using $\mathsf{Com}$. Note that since $A$ does not have the secret key or the randomness used to generate $(\mathsf{sk}_f, \mathsf{vk}_f)$, it commits to the 0 string as randomness. It then emulates the protocol honestly. In either of the signature slots, whenever $A$ needs to provide a signature under $\mathsf{sk}_f$ for any message $m$, $A$ queries its oracle on $m$. For all the other keys, it possesses the signing key and can generate signatures on its own. If the sets $J_1$ or $J_2$ revealed in $P_{1.2}$ and $P_{4.2}$ contains $f$, then $A$ simply halts.
  (2) If $P^*$ fails to convince the verifier in the selected session, $A$ halts. If it succeeds, then $A$ stalls the emulation. Let $C_0$ contain the messages exchanged in session $i$. $A$ then rewinds the prover to the message $V_3$. Let $\tau$ be the partial transcript of the messages exchanged until message $V_3$ in session $i$ occurs.
  (3) Next, $A$ uses fresh randomness to generate $V_3$, namely, the PCPP queries [6]. It then continues the execution from $\tau$ until $P^*$ either convinces the verifier

[6] Here we use the reverse sampling property of the underlying PCPP of proximity.

in that session or aborts. If it aborts then $A$ halts. Let $C_1$ be the transcript obtained from the second continuation of $\tau$.

(4) Using the values revealed by $P^*$ in the two continuations from the point $\tau$, $A$ will try to extract a collision if one exists and halts otherwise.

We describe below how the adversary $A$ obtains a collision from two convincing transcripts starting from $\tau$ and argue that it can do so with non-negligible probability. Thus, we arrive at a contradiction to collision-resistance property of the signature scheme and will conclude the proof of resettable-soundness.

First, we consider a hybrid experiment HYB$'$, where a hybrid adversary $A$ is provided with the actual commitment $c$ to the randomness used to generate the signing key whose signing oracle it has access to. Besides that $A'$ proceeds identically to $A$. By construction the internal emulation by $A$ in HYB$'$ is identical to that of HYB. Below we analyze $A$'s success in hybrid experiment HYB$'$. Finally, we claim that $A$ will succeed with probability close to hybrid experiment HYB$'$ because the commitment scheme is hiding.

In a convincing session, for every PCPP query and every unopened signature key $f$, there is an associated set of paths that the prover reveals. More precisely, for every node $\gamma$ in the paths, the prover provides $encode^\gamma$ which is the vector of commitments: $\{\mathsf{Com}(\mathsf{S}^\gamma[i])\}_{i\in[n]}$ of shares, which are supposed to reconstruct to valid signatures. If a node $\gamma$ is supposed to be the child of the node $\gamma'$, then it must be the case that $label^{\gamma'}$ contains the valid signatures of $\{\mathsf{Com}(\mathsf{S}^\gamma[i])\}_{i\in[n]}$.

Suppose that, in two convincing continuations from $\tau$, for some pair of parent and child node $\gamma'$ and $\gamma$, $label^{\gamma'}$ associated with $\gamma'$ is the same in both the continuations but the commitments in the *encode* part of $\gamma$ are different. This means that there is at least one signature in the $label^{\gamma'}$ that verified correctly on two different values of $encode^\gamma$. Therefore, if $A$ finds one such pair of nodes for the key $\mathsf{sk}_f$, then it obtains a collision. We show below that with non-negligible probability, $A$ will obtain a collision in this manner.

Next, we analyze the set of bad events when $A$ receives two convincing continuations from $\tau$. These bad events prevent $A$ from finding a collision. So we bound the probabilities of these events happening. Fix a $\tau$ for which a random continuation yields a convincing session $i$ with non-negligible probability.

$B_1$**:** $P^*$ **equivocates the commitments.** If $P^*$ equivocates the commitments then it can always compute $t$ accepting views on the fly (as the honest prover does), and $A$'s strategy fails. However, given that $x_n \notin L$, the commitment scheme used in the construction is statistically binding. Thus this event can happen only with negligible probability.

$B_2$**:** $P^*$ **commits to a machine** $M$ **that predicts** $V$**'s next message.** Here $P^*$ computes consistent trees and convinces the verifier using the same algorithm of the simulator. However, because the string $r$ is chosen at random, the probability that this case happens is close to $2^{-n}$, therefore is negligible.

$B_3$**:** $P^*$ **cheats in the proofs of consistency.** Given that $x_n \notin L$, in step $P_5$ the prover convinces the verifier by proving that there exists an index $J$ in which it constructed a consistent tree. $P$ convinces the verifier by running MPC-in-the-head that proves the consistency of nodes and that the leaves are

accepted by a PCPP verifier. Since $B_1, B_2$ happen with negligible probability, $P^*$ cannot convince the verifier by neither equivocating the commitment nor by using a legitimate witness. Thus, it must be the case that $P^*$ convinces the verifier computing an accepting MPC-in-the-head for a false predicate (i.e., the nodes are not consistent/not accepting). However, due to the $t$-robustness of the MPC protocols, this event happens with negligible probability.

Let $J, J'$ be the coordinates that are chosen by $P^*$ in the first and in the second continuation from $\tau$ generated by $A$. (Recall that $J$ is the coordinate for which $P^*$ will be required to provide an accepting proof. For any other coordinate $P^*$ is not required to construct an accepting tree or provide any proof.) We now estimate the probability that $J = J' = f$, where $f$ is the coordinate chosen by $A$. Let $p$ be the probability that for a random continuation from $\tau$, $B_1$, $B_2$ and $B_3$ do not occur. Since $C_0$ and $C_1$ are random continuations, it holds with probability at least $p^2$ that $J = J'$. The index $f$ is chosen uniformly at random by $A$ and is completely hidden. Hence with probability at least $\frac{p^2}{n}$, $J = J' = f$. Whenever this happens, we claim that $A$ can find two nodes $\gamma'$ and $\gamma$ that will yield two strings with the same signature under key $\mathsf{sk}_f$. This is because from $B_2$ we know that $P^*$ cannot compute an accepting PCPP (as the simulator) and from $B_3$ we know that it cannot cheat in the proof. Hence, following similar arguments as in [2,16], we have that a prover $P^*$ that convinces $V$ must be able to open a random leaf of the tree as the value 0 and 1 with non-negligible probability. Given that the root and depth of the PCPP are fixed in $\tau$, we have that in two random continuations $C_0, C_1$ there is a non negligible probability that $P^*$ opens the same leaf as two different values. If this event happens that it must be the case that $P^*$ has found a collision (conditioned on $B_1, B_2, B_3$ not occurring). Thus, $A$ will find a collision with a polynomially related probability therefore contradicting the collision-resistance property of the underlying signature scheme.

**Proof Sketch of Argument of Knowledge:** Proving argument of knowledge will essentially follow from the same approach as the proof of soundness. Assume $P^*$ convinces a verifier on a statement $x$. In the soundness proof, we crucially relied on the fact that the prover cannot equivocate any of the commitments and they were statistically binding. While proving argument of knowledge, this does not hold as $x$ is in $L$. Instead, we observe that if the commitments were binding then $P^*$ can find collisions to the signature scheme and that will happen only with negligible probability. Hence, it must be the case that $P^*$ is equivocating the commitments. Since the commitments are instance based-commitments, if a commitment is opened to two different values, a witness can be extracted. The only witness for all the instance-based commitments used in the protocol is the witness to the input statement $x$. Since $P^*$ equivocates with non-negligible probability, we can extract a witness with non-negligible probability. The extractor essentially runs the same algorithm as $A$ does from the proof of soundness with the exception that it tries to find equivocated commitments as opposed to collision in the signatures.

**Zero-Knowledge.** The simulator Sim internally emulates the code of $V^*$ and proceeds as follows: Sim proceeds as the honest prover through the first signature slot until $V^*$ reveals the keys and randomness in $J_1$. If the set of signatures are not valid or consistent with what is revealed then the simulation halts immediately and outputs the transcript up to that point. Otherwise, Sim stalls the current execution and repetitively queries $V^*$ at the Signature Slot 1 with fresh commitments to $0^v$ (where $v$ is the size of a VSS share), until it obtains $2n$ sets of valid signatures (i.e. $2n$ rewindings where the verifier did not abort in the signing slot or return any invalid signature). Let $\theta$ be the number of queries Sim makes. If $\theta \geq 2^{n/2}$, then Sim aborts outputting $\mathsf{fail}_1$. Next, it sets up a sub-procedure GetSign which is described below: Let $c$ be a commitment to a message $m \in \{0,1\}^v$ and $\kappa \in [n]$.
**Procedure** $\mathsf{GetSign}(c, m, \kappa)$: Initialize $SIGS = \phi$ to be an empty list.
For $i = 1$ to $2n^2\theta T^2$

> Choose $i$ uniformly at random from $[T]$. Select $T-1$ commitments to $m$, $c_1, \ldots, c_{i-1}$, $c_{i+1}, \ldots c_T$ where each of them are obtained by running $\mathsf{Com}^{\mathsf{x}}$ on message $m$. Set $c_i = c$. Rewind $V^*$ to message $P_1$ and feed $(c_1, \ldots, c_T)$ (the first Signature Slot). If $V^*$ replies a valid set of signatures $(\sigma'_1, \ldots, \sigma'_T)$ under verification key $\mathsf{vk}_\kappa$, then $\sigma'_i$ is a valid signature of $c_i = c$. Add $\sigma'_i$ to the list $SIGS$.

If fewer than $n$ signatures are in the list $SIGS$, then return $\perp$. Let $\sigma^*$ be that signature that occurs more than half the time in $SIGS$. Return $\sigma^*$. If no such signature exists return $\perp$.

Next, the simulator proceeds from where it stalled. It creates an (extendable) Merkle signature-tree for $\mathsf{ECC}(V^*)$ for every unopened signing key $\kappa \in [n]/J_1$. To obtain signature on a message $m$ (to be used in the tree construction), it calls $\mathsf{GetSign}(c, m, \kappa)$ procedure $T$ times, each time with a fresh commitment $c$ to $m$. If any of the $T$ trials succeed then it uses that commitment and signature for $m$. For every key $\kappa \in [n]/J_1$ it feeds the values stored in the encode part of the root to $V^*$. If for any message $m$ and key $\kappa \in [n]/J_1$, GetSign fails to return a signature in $T$ tries, then the simulator adds $\kappa$ to the list BadKeys and just commits to dummy values for the roots of those trees by generating VSS shares of the all 0-string. It also commits to the depth of the tree, i.e. $\log|V^*|$ by generating its VSS and committing to each share. Sim receives the challenge $r$ from $V^*$ and proceeds similarly to generate and commit to the root and depth of the (extendable) Merkle-signature tree corresponding to the PCP of Proximity proof and as before, it appends BadKeys with keys for which the simualator fails to obtain the required signatures. After receiving the randomness required to generate the PCPP queries from $V^*$, Sim prepares the responses to get signatures in the second slot. For the keys $\kappa \in \mathsf{BadKeys}$, the simulator simply presents commitments to $0^v$. For every $\kappa \in [n]/(J_1 \cup J_2 \cup \mathsf{BadKeys})$, the simulator proceeds as follows. Let $N$ be the total number of commitments that it has to send for every key. Denote the commitments by $c_1, \ldots, c_N$. Define $\mathsf{VS} \subset [N]$ to be the subset of indexes that contain the commitments that the verifier expects to see

for $q_{i^*,j^*}$ where $i^*, j^*$ are the actual depths of the two trees. The commitments for a key $\kappa$ are generated as follows:

**GENERATE($\kappa$):** $\mathsf{Sim}$ first generates $\tilde{c}_1, \ldots, \tilde{c}_N$ where each $\tilde{c}_i$ is a commitment to $0^v$ using $\mathsf{Com}^{\mathsf{x}}$. For every $\beta \in \mathsf{VS}$, run $\mathsf{GetSign}(\tilde{c}_\beta, 0^v, \kappa)$ and see if it returns a valid signature. If it receives a signature for all commitments, then replace all commitments $\tilde{c}_\beta$ by $c'_\beta$ for all $\beta \in \mathsf{VS}$. Return $\tilde{c}_1, \ldots, \tilde{c}_N$ to be used for the key $\kappa$ in the second slot. If for some commitment a signature was not obtained we say that GENERATE failed for $\kappa$ and add $\kappa$ to $\mathsf{BadKeys}$.

After receiving the signatures, if for some key the signatures obtained for the commitments in $\mathsf{VS}$ were different in the first and second slot, the key is added to $\mathsf{BadKeys}$. If $\mathsf{BadKeys}$ contains all keys not in $J_1 \cup J_2$, i.e. $[n] = \mathsf{BadKeys} \cup \mathsf{J}_1 \cup \mathsf{J}_2$, the simulator halts outputting $\mathsf{fail}_2$. Otherwise, $\mathsf{Sim}$ proceeds to complete the execution by using some key $\kappa^* \in [n]/(J_1 \cup J_2 \cup \mathsf{BadKeys})$ to complete the simulation.

**Running Time of the Simulator.** First we analyze the running time of $\mathsf{Sim}$, Let $p(m)$ be the probability that $\tilde{V}^*$ on query a random commitment $c = \mathsf{Com}(m, \tau)$ of $m \in \{0,1\}^l$ at the Signature Slot 1, returns a valid signature of $c$. Let $p = p(0^l)$.

We first argue that the simulator runs in expected polynomial time. To start, note that $\mathsf{Sim}$ aborts at the end of the Signature Slot 1 with probability $1-p$, and in this case, $\mathsf{Sim}$ runs in polynomial time. With probability $p$, $\mathsf{Sim}$ emulates $V^*$ only a strictly polynomial number of times and size of $V^*$ is bounded by $T_{\tilde{V}^*}$. Thus, $\mathsf{Sim}$ runs in some $T' = poly(T_{\tilde{V}^*})$ time and makes at most $T$ queries to its $\mathsf{GetSign}$ procedure, which in turn runs in time $\theta \cdot poly(n)$ to answer each query. Also note that $\mathsf{Sim}$ runs in time at most $2^n$, since $\mathsf{Sim}$ aborts when $\theta \geq 2^{n/2}$. Now, we claim that $\theta \leq 10n/p$ with probability at least $1 - 2^{-n}$, and thus the expected running time of $\mathsf{Sim}$ is at most

$$(1-p) \cdot poly(n) + p \cdot T' \cdot (10n/p) \cdot poly(n) + 2^{-n} \cdot 2^n \leq poly(T_{\tilde{V}^*}, n).$$

To see that $\theta \leq 10n/p$ with overwhelming probability, let $X_1, \ldots, X_{10n/p}$ be i.i.d. indicator variables on the event that $V^*$ returns valid signatures for a random commitments to $0^s$. If $\theta \leq 10n/p$ then via a standard Chernoff bound, we can conclude that $\sum_i X_i \leq 2n$ happens with probability at most $2^{-n}$. Using a Markov argument, this also proves that the probability of $\mathsf{fail}_1$ occurring is negligible.

**Indistinguishability of Simulation.** To prove indistinguishability, we analyze a hybrid simulator that has the witness and proceeds exactly as $\mathsf{Sim}$ with the exception that for every commitment it uses the equivocal commitment $\mathsf{EQCom}^{\mathsf{x}}$ scheme instead of $\mathsf{Com}^{\mathsf{x}}$. Indistinguishability of the simulation will follow using a standard hybrid argument and the indistinsguishability of the commitment scheme (analogous to [11]). Conditioned on the hybrid simulator not ending in one of the $\mathsf{fail}$ events, we can argue that the output of the hybrid simulator is identical to the real view. Notice that all messages until the second slot will be prepared by the simulator identical to the real simulator. Recall that the messages in the second signature slot are replaced with good commitments according

to the GENERATE procedure. However, since we are replacing one random good commitment with another, the distribution of the simulator will be identical to the real provers message. Finally the rest of the messages only reveal $t$-views of all the MPC protocols and by the perfect $t$-privacy of the MPC protocols these messages will also be identically distributed. Therefore to prove correctness, it suffices to argue that all the $\mathsf{fail}$ events occur with negligible probability.

*Claim.* Except with negligible probability, there are at least $n/2 - n/10$ keys in $[n]/J_1$ for which the hybrid simulator obtains the deterministically signed signatures with probability at least $1 - \log^2 /n$.

*Proof.* Let $s = \frac{\log^2 n}{2}$. For every key, we define a good set of random tapes $G_n$. Now, we say that a key is good if $\mathsf{GetSign}$ fails to return a signature for at most $2s/n$ fraction of the good tapes. First, we show that on a good tape the $\mathsf{GetSign}$ procedure obtains $n$ signatures with high-probability. Next, we show that the probability of a random tape being good is at least $1 - n/T$. Recall that it returns a signature only if it obtains $n$ signatures and there exists a majority. We show that there exists at least $n/2 - n/10$ keys that are good. Suppose these claims were true, then it holds that there are at least $n/2 - n/10$ keys for which the probability that a random tape is good with probability least $1 - n/T - s/n = 1 - 2s/n$. Since the simulator calls $\mathsf{GetSign}$ $T$ times for any message, it will yield a signature for every message in the good $n/2 - n/10$ keys with high-probability.

**Defining $G_n$.** Recall that, in the run-time analysis we showed that $n/p \leq \theta \leq 2^{n/2}$ with probability at least $1 - 2^{-\Omega(n)}$. Now, for every $m \in \{0,1\}^s$, $p(m) \geq p - \nu \geq p/2$ implies that $\theta \geq n/2p(m)$. Fix a message $m$ and a key. Define $G_n$ to be the set of random tapes $\tau$ such that the probability that $V^*$ returns a signature on $(c, c_{-i})$ where $i \leftarrow [n]$, $c = \mathsf{Com}^{\mathrm{x}}(m;\tau)$ and $c_{-i}$ are random $T-1$ commitments for $m$ is at least $\frac{1}{2\theta T^2}$. This means that, for any $\tau \in G_n$, in $2n\theta T^2$ tries, the probability that $\mathsf{GetSign}$ fails to return a single signature is at most $e^{-n}$. Since $\mathsf{GetSign}$ makes $n(2n\theta T^2)$ attempts, it obtains $n$ signatures except with negligible probability.

**Probability of a Good Tape.** We argue that a random $\tau$ is in $G_n$ with probability at least $1 - \frac{n}{T}$. Assume for contradiction the fraction of tapes in $G_n$ was smaller than $1 - \frac{n}{T}$. We now estimate the probability that $V^*$ returns a signature on random commitments. There are two cases: (1) At least one commitment among the $T$ commitments is not in $G_n$. Conditioned on this event, the probability that $V^*$ honestly provides signatures is at most $\frac{T^2}{2\theta T^2}$. (2) All commitments are in $G_n$. The probability this occurs is at most $(1 - \frac{n}{T})^T \leq e^{-n}$. Overall the probability that $V^*$ answers is at most $\frac{1}{2t} + e^{-n} < \frac{1}{\theta} < p(m)$ which is a contradiction. Therefore $G_n$ must contain at least $1 - \frac{n}{T}$ fraction of the tapes.

**Number of Good Keys.** We need to argue that for most messages there will be a majority when $n$ signatures are obtained. We will show that for most messages the $n$ signatures have a majority and the popular signature will be the one that is computed deterministically. At the end of the first signature slot, the verifier

opens the randomness and signing key used in the half the coordinates, i.e. those in $J_1$. Since $J_1$ is committed using an equivocal commitment in this hybrid, it is statistically hiding. So, given the commitments sent by the prover, $J_1$ is completely hidden in Hybrid $H_2$. Therefore, we can conclude that the probability that the verifier gives signatures that were not deterministically signed by more than $s$ keys is at most $2^{-O(s)}$. Using an averaging argument, it holds that the probability that there exists more than $n/10$ keys such that the probability that the verifier gives "incorrect" signatures in those coordinates with probability bigger than $\frac{s}{10n}$ over messages sent in $P_1$ is negligible. This means that, for the remaining $n/2 - n/10$ keys, at most $s/10n$ fraction of possible messages (in the first slot) yield incorrect signatures. We argue next that GetSign gives the wrong signature for a random commitment to a message $m$ in any of these $n/2 - n/10$ keys with probability at most $s/n$. Assuming this holds, it holds that for any message $m$, the probability that GetSign returns a deterministically signed signature for any of the $n/2-n/10$ keys is at least $1-\frac{s}{n}$. We now proceed to prove this claim.

The intuition is that if fewer than $1-s/n$ fraction of the commitments yielded the correct signature with majority, then there will only be a small fraction of $T$-tuple of messages containing such commitments. Recall that, at least $1 - s/10n$ fraction of all $T$-tuple of messages are signed deterministically. Hence with probability at least $1-s/n$ more than half of the $T$ commitments must be deterministically signed and these commitments yield correct signatures with majority. Now suppose that fewer than $1-s/n$ fraction of commitments yielded the deterministically generated signatures in majority. Then it must hold that $(1 - s/n)^{T/2} \geq (1 - s/n)p(m)$. Since $T = O(n^c)$, we can set $c$ sufficiently large ($c = 5$ will suffice) to arrive at a contradiction since $p(m) > \frac{n}{2^{n/2}}$. This concludes proof of the claim.

*Claim.* The probability that the simulator outputs $\mathsf{fail}_3$ is negligible.

*Proof.* We need to show that after the second signature slot BadKeys does not contain all the keys in $[n]/J_1 \cup J_2$. First we show that, GENERATE swaps the commitments for at least $n/20$ keys in $[n]/J_1 \cup J_2$. For the particular depths $i^*, j^*$, at most $\log^e(n)$ ($= |\mathrm{VS}|$) commitments are sent (for some constant $e$). From the previous claim, we know that for at least $n/2 - n/10$ keys in $[n]/J_1$, (and therefore $n/4 - n/10$ keys in $[n]/J_1 \cup J_2$) , GetSign returns a signature on a commitment with probability at least $1 - \log^2 n/n$. A key fails in GENERATE if for some commitment among the commitments with index in $VS$ does not yield a signature through GetSign. This happens with probability at most $\frac{\log^{e+2}(n)}{n}$. Since there are $n/4-n/10$ keys in $[n]/J_1 \cup J_2$, the probability that more than $n/10$ keys fail in GENERATE is $(\frac{\log^{e+2}(n)}{n})^{O(n)}$, i.e. negligible. This means that for at least $n/20$ keys GENERATE successfully swaps. Next, we need to show that there exists at least one key for which the signature obtained by the simulator in the first and second are the same. Recall that GetSign in these keys returns the deterministic signatures on $1 - \log^2 n/n$ fraction of the commitments. Since we are concerned only about $\log^e(n)$ commitments, using the same argument, we can conclude that, the probability that there are more than $n/40$ keys for which

the signature of some commitment among the $\log^e(n)$ commitments obtained in the first slot is not the one deterministically signed is negligible. In other words, there must be at least $n/20 - n/40 = n/40$ keys in $[n]/J_1 \cup J_2 \cup \mathsf{BadKeys}$ for which the simulator obtained deterministic signatures for the $\log^e(n)$ commitments in the first slot and GENERATE inserted those commitments in the second slot. Finally, from the second cut-and-choose, it will follow that there is at least one key among those for which it receives a deterministically signed signature for all the $\log^e(n)$ commitments in the second slot, and hence the same signature obtained from the first slot.

**Acknowledgments.** We thank the anonymous FOCS's reviewers for pointing out an issue with using digital signatures based on one-way functions in a previous version of our work. We thank Kai-Min Chung, Vipul Goyal, Huijia (Rachel) Lin, Rafael Pass and Ivan Visconti for valuable discussions.

## References

1. Barak, B.: How to go beyond the black-box simulation barrier. In: FOCS, pp. 106–115. IEEE Computer Society (2001)
2. Barak, B., Goldreich, O.: Universal arguments and their applications. In: Computational Complexity, pp. 162–171 (2002)
3. Barak, B., Goldreich, O., Goldwasser, S., Lindell, Y.: Resettably-sound zero-knowledge and its applications. In: FOCS 2001, pp. 116–125 (2001)
4. Ben-Sasson, E., Goldreich, O., Harsha, P., Sudan, M., Vadhan, S.P.: Robust pcps of proximity, shorter pcps, and applications to coding. SIAM J. Comput. 36(4), 889–974 (2006)
5. Bitansky, N., Paneth, O.: On the impossibility of approximate obfuscation and applications to resettable cryptography. In: STOC, pp. 241–250 (2013)
6. Choi, S.G., Dachman-Soled, D., Malkin, T., Wee, H.: Simple, black-box constructions of adaptively secure protocols. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 387–402. Springer, Heidelberg (2009)
7. Chor, B., Goldwasser, S., Micali, S., Awerbuch, B.: Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults (Extended Abstract). In: Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1985, pp. 383–395 (1985)
8. Chung, K.M., Ostrovsky, R., Pass, R., Venkitasubramaniam, M., Visconti, I.: 4-round resettably-sound zero knowledge. In: TCC. pp. 192–216 (2014)
9. Chung, K.M., Pass, R., Seth, K.: Non-black-box simulation from one-way functions and applications to resettable security. In: STOC (2013)
10. Dachman-Soled, D., Kalai, Y.T.: Securing circuits against constant-rate tampering. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 533–551. Springer, Heidelberg (2012)
11. Dachman-Soled, D., Malkin, T., Raykova, M., Venkitasubramaniam, M.: Adaptive and concurrent secure computation from new adaptive, non-malleable commitments. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part I. LNCS, vol. 8269, pp. 316–336. Springer, Heidelberg (2013)
12. Goldreich, O.: Foundations of Cryptography — Basic Tools. Cambridge University Press (2001)

13. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract). In: STOC, pp. 291–304 (1985)
14. Goyal, V.: Constant round non-malleable protocols using one way functions. In: Fortnow, L., Vadhan, S.P. (eds.) STOC, pp. 695–704. ACM (2011)
15. Goyal, V., Lee, C.K., Ostrovsky, R., Visconti, I.: Constructing non-malleable commitments: A black-box approach. In: FOCS, pp. 51–60. IEEE Computer Society (2012)
16. Goyal, V., Ostrovsky, R., Scafuro, A., Visconti, I.: Black-box non-black-box zero knowledge. In: STOC (2014)
17. Haitner, I.: Semi-honest to malicious oblivious transfer—the black-box way. In: Canetti, R. (ed.) TCC 2008. LNCS, vol. 4948, pp. 412–426. Springer, Heidelberg (2008)
18. Impagliazzo, R., Rudich, S.: Limits on the provable consequences of one-way permutations. In: Goldwasser, S. (ed.) Advances in Cryptology - CRYPT0 1988. LNCS, vol. 403, pp. 8–26. Springer, Heidelberg (1990)
19. Ishai, Y., Kushilevitz, E., Lindell, Y., Petrank, E.: Black-box constructions for secure computation. In: Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, pp. 99–108. ACM (2006)
20. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: Johnson, D.S., Feige, U. (eds.) STOC, pp. 21–30. ACM (2007)
21. Kilian, J.: A note on efficient zero-knowledge proofs and arguments (extended abstract). In: Kosaraju, S.R., Fellows, M., Wigderson, A., Ellis, J.A. (eds.) STOC, pp. 723–732. ACM (1992)
22. Lin, H., Pass, R.: Black-box constructions of composable protocols without set-up. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 461–478. Springer, Heidelberg (2012)
23. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
24. Naor, M., Yung, M.: Universal one-way hash functions and their cryptographic applications. In: STOC 1989, pp. 33–43 (1989)
25. Ostrovsky, R., Scafuro, A., Venkitasubramaniam, M.: Resettably sound zero-knoweldge arguments from owfs - the (semi) black-box way. Cryptology ePrint Archive, Report 2014/284 (2014), `http://eprint.iacr.org/`
26. Pass, R., Rosen, A.: New and improved constructions of non-malleable cryptographic protocols. In: STOC 2005, pp. 533–542 (2005)
27. Pass, R., Tseng, W.-L.D., Wikström, D.: On the composition of public-coin zero-knowledge protocols. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 160–176. Springer, Heidelberg (2009)
28. Pass, R., Wee, H.: Black-box constructions of two-party protocols from one-way functions. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 403–418. Springer, Heidelberg (2009)
29. Reingold, O., Trevisan, L., Vadhan, S.P.: Notions of reducibility between cryptographic primitives. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 1–20. Springer, Heidelberg (2004)
30. Rompel, J.: One-way functions are necessary and sufficient for secure signatures. In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, Baltimore, Maryland, USA, May 13-17, pp. 387–394. ACM (1990)
31. Wee, H.: Black-box, round-efficient secure computation via non-malleability amplification. In: FOCS, pp. 531–540. IEEE Computer Society (2010)