CSAR: A Practical and Provable Technique to Make Randomized Systems Accountable

Michael Backes[1,2] Peter Druschel[2] Andreas Haeberlen[2,3] Dominique Unruh[1]

[1] Saarland University [2] MPI-SWS [3] Rice University

Technical Report MPI-SWS-2008-002

Abstract

We describe CSAR, a novel technique for generating cryptographically strong, accountable randomness. Using CSAR, we can generate a pseudo-random sequence and a proof that the elements of this sequence up to a given point have been correctly generated, while future values in the sequence remain unpredictable. CSAR enables accountability for distributed systems that use randomized protocols. External auditors can check if a node has deviated from its expected behavior without learning anything about the node's future random choices. In particular, an accountable node does not need to leak secrets that would make its future actions predictable. We demonstrate that CSAR is practical and efficient, and we apply it to implement accountability for a server that uses random sampling for billing purposes.

1 Introduction

Nodes in distributed systems can fail for many reasons: a node can suffer a hardware or software failure, an attacker can compromise a node, or a node's operator can deliberately tamper with its software. Moreover, faulty nodes are not uncommon [24]. As a system grows larger, it is increasingly likely that some nodes are accidentally misconfigured or have been compromised as a result of unpatched security vulnerabilities.

Recent work has explored the use of *accountability* to detect and expose node faults in distributed systems [29, 15]. Accountable systems maintain a tamper-evident record that provides non-repudiable evidence of all nodes' actions. Based on this record, a faulty node whose observable behavior deviates from that of a correct node can eventually be detected. At the same time, a correct node can defend itself against any false accusations.

In PeerReview [15], for instance, each node maintains a tamper-evident log, which records all messages the node sends and receives as well as inputs and outputs of the application. Any node i can request the log of another node j and independently determine whether j has deviated from its expected behavior. To do this, i replays j's log using a reference implementation that defines j's expected behavior. By comparing the results of the replayed execution with those recorded in the log, PeerReview can detect observable Byzantine faults without requiring a formal specification of the system.

The approach taken by PeerReview is very general, but it requires that each node's actions be deterministic; otherwise, a different non-deterministic choice by a node and its reference implementation would be classified incorrectly as a fault. One approach to ensure deterministic behavior is to disclose, as part of a node's record, the seed of any pseudo-random number generator used in the node's program. Unfortunately, disclosing the seed also reveals any secrets that were randomly chosen by the node and makes the future sequence of pseudo-random numbers predictable. One could allow a node to choose a new seed once it has proven that its past actions were fault-free. However, this would allow a bad node to choose seeds strategically, and thus to influence its own pseudo-random numbers.

Thus, applying existing accountability techniques faces us with a choice: we can make a node's actions (including its adherence to a pseudo-random sequence) accountable at the expense of revealing the node's secrets and making its future actions predictable; or, we can protect a node's secrets and keep its future actions unpredictable, but give up the ability to verify that the node is following a pseudo-random sequence of actions.

Consider, for instance, a distributed algorithm that uses some form of statistical sampling. We would like to be sure that each node follows a truly random sequence of samples to ensure unbiased results. However, disclosing a node's future random samples as a side-effect of auditing the node's past actions may allow an attacker to adapt his behavior to the expected sampling, thus biasing the results. As a result, existing accountability techniques are not appropriate for such protocols.

1.1 Our contributions

We contribute CSAR, a technique for generating Cryptographically Strong, Accountable Randomness. CSAR allows us to apply accountability techniques to probabilistic protocols without making their actions predictable. More precisely, we propose a pseudo-random generator that satisfies the following requirements:

1. The pseudo-random generator should output cryptographically strong randomness. It is not sufficient for the output of the generator to be uniformly distributed. We require that the node generating the output should only be able to compute values it could also compute if the output was truly random.[1]

2. The pseudo-random generator should be accountable, i.e., after each random value r is generated, it should be possible to generate a proof that this value r was indeed correctly derived from a given seed. Thus, if a node generates a value incorrectly, it can be held accountable because it cannot produce a valid proof.

3. Future random values of correct nodes should be unpredictable, i.e., to a node that learns random values $r_1, \dots, r_i$ and the corresponding proofs, all future random values $r_{i+1}, \dots$ should still look random. This excludes the obvious solution of using the random seed as a proof.

4. Properties 1-3 should hold even if malicious nodes are present while the seed is computed. In particular, no node should be able to influence the output of its own generator by choosing a suitable seed.

Additionally, both generating the randomness and verifying the corresponding proofs should be highly efficient, in order to limit the cost of accountability relative to the actual protocol execution. This requirement excludes a general solution based on zero-knowledge proofs.

CSAR achieves these goals with a protocol in which an initial coin-toss is followed by a combination of hashing (where the hash function is modeled as a random oracle) and a trapdoor one-way permutation. Our construction essentially constitutes a chain of inverse trapdoor applications starting from the seed derived from the coin-toss, where the sequence is partitioned into blocks by intermediate applications of the hash function. The hash function is additionally used to transform elements of this sequence into independent random values. The overall construction resembles existing techniques for generating keys in cryptographic file systems, e.g., [16, 1]. Elements in the sequence serve as a proof for former sequence elements and hence for the corresponding random values, since a third party can use the permutation to compute former sequence values and compare them with the random values that were used. The hardness of inverting the trapdoor permutation and the usage of the random oracle prevent a prediction of future sequence elements. This construction is efficient (requiring only a few hashes and multiplications in an RSA group for each generation of a random value), and it can be further optimized by exploiting number-theoretic properties of low-exponent RSA.

The security of CSAR is formally established by comparing it to an ideal specification of its expected behavior, under the additional hypothesis that the surrounding protocol does not use the same hash function as that used for generating the randomness. This corresponds to the well-known simulatability paradigm of modern cryptography. Among these, the Reactive Simulatability (RSIM) framework [3] and the Universal Composability (UC) framework [8] constitute the most prominent representatives; they have been used to prove the security of various protocols. In particular, simulatability offers strong compositionality guarantees.

CSAR can be used with different accountability techniques; however, for concreteness, we present it in the context of PeerReview. We implemented CSAR as an extension to the publicly available PeerReview library [25]. Adding support for accountable randomness enables the use of PeerReview in applications that rely on unpredictable random choices. Such applications include, for instance, systems that rely on random sampling for security monitoring or billing, randomized load balancing in federated systems or randomized replica placement in distributed storage systems. Our evaluation shows that the computational cost of our technique is low: on current hardware and with a 1024-bit RSA modulus, a random number can be generated in less than $20\mu s$ and verified in less than $10\mu s$. We also show that CSAR is practical and that its storage and bandwidth costs are low, both in relative and in absolute terms.

1.2 Related work

Verified random functions (VRFs) [22] and the stronger simulatable VRFs [11] are closely related to the technique proposed in this paper. However, even simulatable VRFs cannot guarantee that the randomness produced by malicious parties has strong properties when the malicious parties release additional information about their seeds; hence simulatable VRFs are not sufficient for the scenario considered in this paper. Furthermore, VRFs, and even more so

[1] As a counterexample, consider a pseudo-random generator that produces random numbers as $r = g^x$ in some group G, where x is a random element. The output of this generator is uniformly distributed, but the node that generates r also knows the discrete logarithm of r - which it could not know if r was a true random number.

simulatable VRFs, are much less efficient than our technique. In CSAR, we obtain the improved efficiency, as well as the ability to produce strong randomness when malicious parties disclose their seeds, by applying the random oracle model, which permits very efficient constructions.

Hash chains [18] can be used to generate verifiable pseudo-random values. However, since each hash chain can produce only a finite number of values, an upper bound on the required output length must be known in advance. Also, the hash chain must either be stored in memory or recalculated from scratch after each invocation, both of which are inefficient. Finally, the initial hash value must remain secret, which enables an attacker to influence at least some bits of his randomness by choosing a suitable initial hash. None of these limitations apply to CSAR.

Accountability in distributed systems has been suggested as a means to achieve practical security [19], to create an incentive for cooperative behavior [13], and even as a general design goal for dependable networked systems [28]. Several recent systems provide accountability for deterministic systems [30, 23, 15]. None of these systems can hold a node accountable for its random choices without also making its future choices predictable, which can make the node vulnerable to attacks and exploits.

1.3 Outline

The remainder of this paper is organized as follows. Section 2 reviews cryptographic preliminaries such as the random oracle model and simulatable security notions. Section 3 defines the security guarantees CSAR is designed to fulfill. Sections 4 and 5 present the protocol for generating accountable randomness and its security proof, respectively. Section 6 sketches the implementation of CSAR in the context of PeerReview, while Section 7 discusses applications of CSAR. Section 8 reports on experimental results to measure the efficiency and storage consumption of CSAR. Section 9 discusses possible variations of our approach, and Section 10 concludes the paper.

2 Preliminaries

2.1 The random oracle model

The random oracle model [5] is one of the most popular heuristics in cryptography. The security of virtually all practically deployed public-key encryption and signature schemes relies on the random oracle model, e.g., that of the RSA-OAEP encryption scheme [6] specified in the PKCS #1 standard [26].

The random oracle model formalizes the intuition that a good cryptographic hash function has essentially no recognizable structure, i.e., the function can be expected to behave like a completely random function. Instead of proving the protocol under consideration with respect to some fixed actual hash function H (e.g., SHA-1), proofs in the random oracle model presuppose a function $H : \{0,1\}^* \to \{0,1\}^l$ that is uniformly chosen from the set of all such functions, i.e., for each value x, the value $H(x)$ constitutes a uniformly chosen value (with two calls to $H(x)$ returning the same value). The security of the protocol under consideration is then proven by granting the protocol oracle-access to H; the implementation, however, uses the concrete hash function. Although (pathological) protocols exist that violate the random oracle heuristics [9], to the best of our knowledge there is no example of a practical protocol that is proven secure within the random oracle model but whose implementation turns out to be insecure when implemented with a sufficiently good cryptographic hash function.

The random oracle model permits very efficient protocol constructions. In addition, the random oracle model has the following advantage in our setting: our randomness generation protocol is only provably secure if it relies on a different hash function than the one used in the application protocol. For an actual hash function, this statement is difficult to formalize properly since the application protocol might only compute parts of the hash function, or the function might be obfuscated. If one relies on the random oracle model, this statement can be naturally formalized by not allowing the application protocol to query the oracle H.

2.2 Low-exponent RSA

In the following sections, we consider the low-exponent RSA permutation $f_n(x) := x^3 \bmod n$, where n is a random RSA-modulus (a product of two random primes p and q of the same length) of some length l with $3 \nmid \varphi(n) = (p-1) \cdot (q-1)$. The low-exponent RSA permutation is a variant of the RSA permutation in which the public exponent e is instantiated as a small fixed number (in our case $e = 3$). It is well known that naively using low-exponent RSA in larger protocols is known to yield troublesome scenarios. For example, using it as an encryption scheme without additional padding allows an adversary to recover a plaintext from seeing three encryptions of this plaintext for three different public keys. However, it is a well-accepted assumption that the low-exponent RSA permutation itself is hard to invert. More exactly, we define the following function $\varepsilon_{3\mathrm{RSA}}$.

Definition 1 *Let $\varepsilon_{3\mathrm{RSA}}(l, s)$ be the maximum probability over all circuits of size at most s that, upon input of a random RSA modulus n of length l and a random $y \in \{0, \ldots, n-1\}$, the circuit outputs some x with $x^3 \equiv y \bmod n$.*

The low-exponent RSA assumption for $e = 3$ (abbreviated 3RSA) can be formally stated as follows:

Assumption 1 (3RSA) *For $l(k) \in \Omega(k)$ and any polynomial s, $\varepsilon_{3\mathrm{RSA}}(l(k), s(k))$ is negligible.*

The 3RSA assumption trivially follows from the well-established strong RSA assumption [4]. In addition, the function f_n can be inverted efficiently if the factorization of $n = pq$ is known: One computes a secret key d with $3d \equiv 1 \bmod \varphi(n)$ and then computes $f_n^{-1}(x) = x^d \bmod n$. In other words, under the 3RSA assumption, f_n constitutes a trapdoor one-way permutation.

2.3 Simulatable security

The security guarantees CSAR is designed to fulfill will be defined by an ideal functionality, which serves as a specification of the protocol's desired behavior. Simulatable security then aims at showing that a protocol is as good as its ideal functionality. This is formalized by requiring that for any adversary A that attacks the protocol (i.e., an adversary that controls the malicious nodes and may intercept information) there exists a simulator S that attacks the ideal functionality of the protocol, such that any third entity, called the environment and intuitively denoting the application built on top of the protocol, cannot distinguish between a run of the real protocol with A and an execution of the ideal functionality with S. This approach for defining properties of cryptographic systems is widely used in the cryptographic community, where it is known as UC security (Universal Composability) [8] or as RSIM security (Reactive Simulatability) [3]; we refer to these papers for the rigorous definitions. These definitions provide very strong security and compositionality guarantees [8, 2]. Compositionality constitutes a particularly important property in our setting since we want to use CSAR within a larger context (with the application protocol and with an accountability technique like PeerReview).

3 Desired security guarantees

We now formally define an ideal functionality that corresponds to the security properties CSAR is supposed to achieve. The ideal functionality is defined as a collection of machines $\tilde{M}_P$, one for every entity P. Phrasing the ideal functionality as a (collection of) machine(s) allows us to meaningfully compare it to real protocols within existing simulatable security models, which are all machine-based.

The behavior of the ideal functionality reflects the security properties informally outlined in Section 1.1. The ideal functionality does not generate randomness according to the protocol description; rather, it chooses truly random values r_i. The ideal functionality moreover ensures that even malicious entities cannot lie about their randomness. However, malicious entities are allowed to predict their *own* future random values even if these values have not yet been used by the protocol; moreover, previously used random values of honest entities are revealed to the adversary. We give these powers to the malicious entities in the ideal model to explicitly model the security requirements that are *not* fulfilled by our construction. Hence, the ideal functionality captures the requirement that, intuitively, the randomness generated by CSAR is as good as true randomness, up to the two imperfections mentioned above. These imperfections can be eliminated if desired, but the cost is a computationally more expensive solution, cf. Section 9.

To model the generation of a single random value in the real protocol, we let the functionality output a triple (r_i, s_i, b_i) to the environment. Here r_i corresponds to the randomness, s_i to the audit information, and b_i is a bit which describes whether the audit information is valid. That is, we assume that in the real protocol, any auditor which sees s_i will immediately compute the corresponding bit b_i and consider this derived bit to be part of the audit information. In the real protocol (assuming that it is secure) the adversary will only have two choices: Either it chooses r_i honestly at random and chooses some auditing information s_i such that $b_i = 1$, or it chooses r_i to its liking, but then it may only produce auditing information s_i such that $b_i = 0$. In other words, while the real protocol cannot be designed to output correct values r_i for malicious entities that deviate arbitrarily from the protocol, we can ensure that incorrect values will fail the respective tests. In the ideal functionality, this observation is reflected in the assumption that the adversary can choose the outcome b_i of the test. If the adversary chooses $b_i = 0$, it may choose the "random" value; if the adversary chooses $b_i = 1$, true randomness is always returned. Furthermore, if the entity is honest, only $b_i = 1$ is allowed (as honest agents will never produce invalid audit information). Our security definition in particular does not require any properties about the s_i (only about the result of the verification of the randomness, which is captured by the value of b_i). Consequently, s_i can be chosen by the adversary even in the case of honest parties (this is a popular way to model nondeterminism in cryptographic protocols).

Definition 2 (Ideal Functionality) *The ideal $\langle$honest$\rangle$ [dishonest] machine $\tilde{M}_P$ for entity P performs the following steps, given security parameters l_1 and t_2:*

- *Before the first activation, $\tilde{M}_P$ initializes an infinite list of values $r_1, r_2, \ldots$ uniformly and independently* *distributed over $\{0,1\}^{l_1}$.*[2] *[All values r_i are made accessible to the adversary, i.e., a query i from the adversary is answered with r_i.]*

- *Upon each activation, the inputs to the machine $\tilde{M}_P$ are forwarded to the adversary.*
- *In $\tilde{M}_P$'s first environment activation, $\tilde{M}_P$ asks the adversary for some values $(n, q_1, \ldots, q_{t_2})$. This tuple $(n, q_1, \ldots, q_{t_2})$ is returned to the environment. The values $n, q_1, \ldots, q_{t_2}$ correspond to values that might be used in the setup phase, in order to establish a common random element.*[3]
- *In $\tilde{M}_P$'s second environment activation, a random $s \in \{0,1\}^{l_1}$ is chosen and returned. The value s is also given to the adversary. (s corresponds to the publicly known seed.)*
- *In each subsequent environment activation (indexed consecutively, starting with $i = 1$), $\tilde{M}_P$ sends r_i to the adversary and asks the adversary for a tuple $(\tilde{r}_i, s_i, b_i)$. ⟨Then $\tilde{M}_P$ returns $(r_i, s_i, 1)$.⟩ [Then M_P returns $(r_i, s_i, 1)$ if $b_i = 1$ and $(\tilde{r}_i, s_i, 0)$ otherwise.]*

We check that each of the intuitive security requirements described in Section 1.1 is implied by this ideal functionality: Property 1 holds because the ideal functionality chooses the random values r_i in a truly random way, even for the malicious parties. Property 2 is satisfied because the ideal function will ensure that $b_i = 0$ unless the adversary uses the honestly generated randomness r_i. Property 3 is ensured because the functionality will reveal the random values r_i corresponding to *honest* parties only when an honest party actually requests them. Until then, they are not accessed by any machine. Property 4 is fulfilled because in the ideal model we have modeled that the seed s is chosen in a truly random fashion by the functionality. This implies that any protocol implementing the functionality also has to choose the seed s in a random fashion, even if malicious parties are involved.

Moreover, the functionality also explicitly models the security imperfections of CSAR: The values r_i of malicious agents are revealed to the adversary in advance. Whenever an honest agent uses a random value, that value r_i is revealed to the adversary (because in the real protocol, it appears in the audit log). Malicious parties can actually use non-random values $\tilde{r}_i$; this is only detected by comparing these values to the audit log. The fact that the ideal functionality has to explicitly model all restrictions of the protocol is considered one of the main advantages of simulatable security notions.

[2] Strictly speaking, the whole infinite list is not initialized at the beginning of the protocol, but is lazily built up whenever a value r_i is required.

[3] This step is needed for technical reasons because otherwise the outputs of the protocol described in the next section would look syntactically different from the outputs of the ideal functionality, which is forbidden by simulatable security definitions.

4 The CSAR protocol

We first explain the concepts we exploit in order to achieve the desired security guarantees. Afterwards, we give the formal description of our protocol for generating accountable randomness.

4.1 Informal overview

4.1.1 Accountability and unpredictability

We first illustrate how we achieve the accountability and the unpredictability of the pseudo-random generator, i.e., properties 2 and 3 from Section 1.1. Suppose P is an entity that needs to generate random values. We assume that there is a trapdoor one-way permutation f whose secret key is known only to P (that is, only P can invert the permutation). For now, we will also assume that there is a well-known random seed s_0; in Section 4.1.3, we describe how this value is generated with an initial coin-toss.

Since P is the only entity that can invert the permutation f, it alone is able to compute elements of the sequence $s_i := f^{-1}(s_{i-1})$. The other entities do not have the secret key of f and therefore cannot compute new elements, even if they already know the old elements $s_0, \ldots, s_{i-1}$. However, all entities can *evaluate* f and can therefore validate a new element s_i by checking whether $f(s_i) = s_{i-1}$ holds true. Since f is a permutation, this check is equivalent to $s_i = f^{-1}(s_{i-1})$. (Our proof additionally ensures that f constitutes a permutation even for incorrectly generated keys, hence ensuring accountability for dishonest parties as well.) Thus, we can achieve accountability for those values (by including all s_i in the audit log), and at the same time, prevent future values from being predicted.

However, directly using the elements s_i as the desired random values r_i is not secure, because there is a strong relationship between s_i and s_{i-1} (one being the image of the other under f), which would not be the case if the values were truly random. To avoid this, we use $r_i := H(s_i)$ as the desired random value. When H is modeled as a random oracle, $H(r_i)$ and $H(r_{i-1})$ are decoupled and become independent, random elements.

4.1.2 Strong cryptographic randomness

Providing strong cryptographic randomness in the sense of property 1 from Section 1.1 is difficult in general. Fortunately, the construction outlined above for computing the values r_i can already be shown to offer strong cryptographic randomness, provided that 1) we model H as a random oracle, and that 2) we make the following change to our construction: We first define a hash function $H^*(x) := H(1,x)\| \ldots \|(t_3, x)$ for a certain parameter t_3. Then the

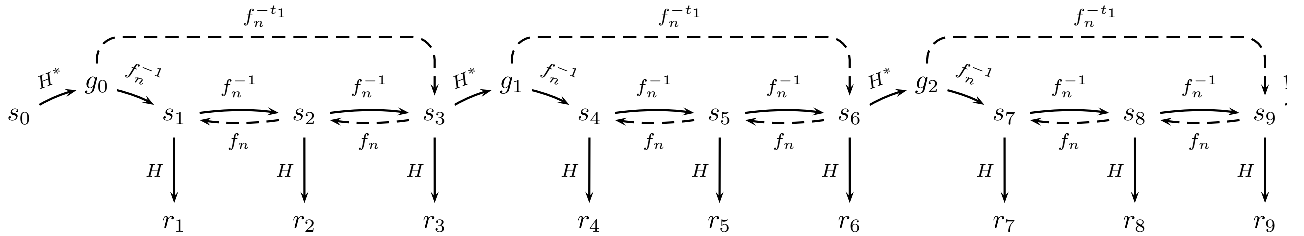


Figure 1: The randomness generator for $t_1 = 3$. The dashed lines depict the optimized variant from Section 6.2.

images of this so-called padded hash function H^* are long enough to be used as arguments to f. Then, in every t_1th step for a parameter t_1, the value s_i is not computed as $s_i = f^{-1}(s_{i-1})$ but as $s_i = f^{-1}(H^*(s_{i-1}))$ (see Figure 1). In the following, we briefly describe how this adaptation enables the security proof in Section 5.

Recall that our work relies on the well-established approach of defining security by means of simulation. To show that a sequence r_1 is random, even given the side information s_i and f (and if P is malicious, additionally the secret key for f), we must show the existence of an efficient machine called the simulator, which, given a sequence of values r_i, can simulate a realistically looking protocol execution that results in exactly these values. In particular, it has to come up with realistic values for s_i and f. Hence, if some property holds for the values r_i in the real protocol, the same property would also hold for the truly random values r_i in the simulation. For instance, if one could compute the discrete logarithm of r_i in the execution of the real protocol, one could also compute the discrete logarithm of the truly random r_i in the simulation, and since the latter is conjectured infeasible, it follows that the discrete logarithm of r_i cannot be computed in the real protocol either – not even by P itself.

In our case, the simulation becomes possible because H is modeled as a random oracle. Since the simulator has to simulate H, it is free to choose the values $H(x)$ in a suitable manner, as long as the distribution of $H(x)$ is still uniform. For example, it can set $H(s_i) := r_i$, provided that it can recognize a value $s = s_i$. The construction of the r_i outlined above does not yet seem to entail an efficient way to recognize such values because arbitrary values s may occur, i might be arbitrarily large, and one would have to test for arbitrarily many i whether $f^i(s) = s_0$ holds. This is why we require the change described earlier, namely that in every t_1-th step, the value s_i is not computed as $s_i = f^{-1}(s_{i-1})$ but as $s_i = f^{-1}(H^*(s_{i-1}))$. Thus, any $s = s_i$ fulfills $f^j(s) = H^*(x)$ for some $j \leq t_1$ and some x. Since the simulator simulates the function H, it knows all values $H^*(x)$ that have been queried from H so far, and thus it can efficiently check whether $f^j(s) = H^*(x)$ holds for some x that has already been queried and for some $j \leq t_1$. For values x that have not been queried, one can easily show that this equation only holds with negligible probability.

In summary, these two modifications allow us to prove that CSAR offers strong cryptographic randomness guarantees, even for randomness produced by malicious entities. We note that $t_1 = 1$ is a perfectly fine choice from a security point of view, but larger values of t_1 can make the implementation more efficient. We describe the details in Section 6.2.

4.1.3 Choosing a suitable seed

We finally turn to the property of suitably choosing the seed, in the sense of property 4 from Section 1.1. So far, our construction presupposed that the initial seed s_0 is chosen randomly, and that the function f is chosen correctly, even if P is malicious. A suitable choice of s_0 can be enforced by choosing s_0 with a coin-toss, which can easily be implemented using the hash function H. Enforcing a correct choice of f turns out to be more difficult. Since the secret key of f must not be disclosed to any participant other than P, P chooses f on its own. This opens the possibility that f could be badly-formed in one of the following two ways.

First, f might not constitute a permutation. In this case, the values s_i will not necessarily be uniformly distributed; worse, some value s_{i-1} may have several preimages s_i under f, so that P may be able to choose the next random value from these possible values. This can be prevented by finding a way to prove that f indeed constitutes a permutation. In particular, this will ensure accountability for dishonest users that might incorrectly generate their keys, but, since the secret key must not be revealed, it is difficult to prove in general. In the case of the low-exponent RSA permutation, however, it turns out to be sufficient to show for a few random values y_i that all these values have a preimage under f. Hence, in order to prove that f constitutes a permutation, CSAR computes values $q_\mu = f^{-1}(H(\mu, n))$, where n is the RSA modulus used by f. We elaborate on this in detail in Appendix A.

The second possibility is that an incorrectly chosen f might have a small period, i.e., for some s_0 and some μ, we might have that $s_{\nu+\mu} = f^\mu(s_\nu) = s_\nu$ and consequently that $r_{\nu+\mu} = r_\nu$. This is circumvented by including P and i in all hash values. Hence, even in the case $s_{\nu+\mu} = s_\nu$, we still have $r_{\nu+\mu} \neq r_\nu$.

4.2 Formal description of CSAR

We now formally describe the protocol for generating accountable randomness. CSAR is designed as a subprotocol for inclusion in some larger application like PeerReview; here, we only specify the routines for generating randomness and for generating and verifying the corresponding proofs. Full-scale accountability is then provided at the next layer, e.g., by PeerReview.

4.2.1 Parameters and additional notation

CSAR is parametrized by the following values: the value l_1 is the length of $H(x)$ for any x. The value l_2 is the length of the RSA modulus used. The values $t_1, t_2, t_3, t_4 \geq 1$ denote integers satisfying $t_3 l_1 \geq l_2$. The security of CSAR will be guaranteed if t_1, t_2, $t_3 l_1 - l_2$, and t_4 are of at least linear size in the security parameter; see also Theorem 1 below. For the setup phase, we additionally need a function ω that maps each entity P to a set of other entities $\omega(P)$ such that at least one entity in each set $\{P\} \cup \omega(P)$ is guaranteed to be honest during the setup phase. The witness set function in PeerReview can be used for this purpose.

We use the following notation: $H(x)$ denotes an application of the random oracle. When writing $H(x, y, \dots)$ we assume that the tuple $(x, y, \dots)$ is encoded into a single string in some efficiently decodable fashion. By $H^*(x)$ we denote $H(1, x) \| \dots \| H(t_3, x)$. Note that the length of $H^*(x)$ is at least l_2. For an integer n (not necessarily an RSA modulus), we write f_n to denote the function $f_n(x) := x^3 \bmod n$. In a slight abuse of notation, we write $f_n^{-1}(x) \in \{0, \dots, n-1\}$ for the preimage of $x \bmod n$ under f_n, provided that f_n constitutes a permutation on $\{0, \dots, n-1\}$. Note though that even if f_n^{-1} is defined, it is the inverse of f_n only on $\{0, \dots, n-1\}$.

4.2.2 Setup phase

CSAR starts with a setup phase for generating the seed and the permutation f. In this phase, each entity P performs the following steps with the entities in $\omega(P)$:

- P chooses a random RSA modulus n such that $3 \nmid \varphi(n)$ and computes the secret key d with $3d \equiv 1 \bmod \varphi(n)$. P does *not* store the secret key in its audit log.
- P computes $q_\mu := f_n^{-1}(H^*(\mathtt{pk}, \mu, n))$ for $\mu = 1, \dots, t_2$ and sends a signed message $(\mathtt{pk}, n, q_1, \dots, q_{t_2})$ to each entity in $\omega(P)$. Here $\mathtt{pk}$ denotes an arbitrary but fixed string that is different from the identifier of any entity.
- The entities in $P \cup \{\omega(P)\}$ perform a coin-toss (see below), which produces a random value s.
- Finally, P sets $s_0 := H^*(P, \mathtt{start}, s)$ where P denotes a string encoding the identity of the entity P, and $\mathtt{start}$ denotes some arbitrary but fixed string that is not an integer.

The setup phase includes a *coin-toss subprotocol* to produce a random value s. Entities $P, P_1, \dots, P_k$ perform a coin toss as follows. First, they choose random values $r, r_1, \dots, r_k$. Then each entity P_i computes $c_i := H(r_i)$ and produces a signature σ_i on c_i. Next, all (c_i, σ_i) are sent to P. P sets $c := H(r)$, $h := (c, c_1, \sigma_1, \dots, c_k, \sigma_k)$, and produces a signature σ on h. Then each P_i checks all signatures in h, produces a signature σ_i' on h, and sends (r_i, σ_i') to P. Finally, P checks all signatures σ_i' and sends $(r, r_1, \dots, r_k)$ to $P_1, \dots, P_k$. The outcome of the coin toss is $s := r \oplus r_1 \oplus \dots \oplus r_k$.

The coin-toss subprotocol can easily be shown to produce a random value s, provided that at least one entity is honest. All messages are signed, so that when plugging the subprotocol into PeerReview, every entity can prove that it indeed behaved correctly (since the coin-toss subprotocol is only invoked once, the communication and computation overhead induced in particular by the signatures is acceptable). We do not require the value s to remain secret; this strongly facilitates performing a secure coin toss, in particular in the random oracle model.

4.2.3 Generating random values

To generate a random value r_i and the corresponding audit information, an entity P performs the following steps. Let i be a sequential index, starting at $i = 1$. If $t_1 \mid i - 1$, P sets $s_i := f_n^{-1}(H^*(P, i-1, s_{i-1}))$; if $t_1 \nmid i - 1$, P sets $s_i := f_n^{-1}(s_{i-1})$. P then chooses $r_i := H(P, i, s_i)$ and stores s_i, r_i in the audit log.

4.2.4 Verifying random values

To verify a random value r_i, an auditor evaluates the following function *Verify* on the values $(P, n, s, r_i, q_1, \dots, q_{t_2}, s_1, \dots, s_i)$, where P is a string encoding the identity of the entity P, s is the value computed in the coin-toss, r_i is the current random value, $q_1, \dots, q_{t_2}$ are the values sent in the setup phase and $s_1, \dots, s_n$ are the values found in the audit log.

Definition 3 (Verification function) *When invoked as* $Verify(P, n, s, r_i, q_1, \ldots, q_{t_2}, s_1, \ldots, s_i)$ *with* $i \geq 1$, *the function Verify performs the following checks:*

- $s_\mu \stackrel{?}{\in} \{0, \ldots, n-1\}$ *for* $\mu = 1, \ldots, i$.
- $f_n(q_\mu) \stackrel{?}{\equiv} H^*(\mathtt{pk}, \mu, n) \bmod n$ *for* $\mu = 1, \ldots, t_2$.
- $f_n(s_\mu) \stackrel{?}{=} s_{\mu-1}$ *for all* $\mu = 1, \ldots, i$ *with* $t_1 \nmid \mu - 1$.
- $f_n(s_\mu) \stackrel{?}{\equiv} H^*(P, \mu - 1, s_{\mu-1}) \bmod n$ *for all* $\mu = 1, \ldots, i$ *with* $t_1 \mid \mu - 1$ *where* $s_0 := H^*(P, \mathtt{start}, s)$.
- $r_i \stackrel{?}{=} H(P, i, s_i \bmod n)$.

An implementation does not need to perform all these checks upon each invocation of *Verify*. Since only one new value s_i occurs for each new randomness query, each evaluation of *Verify* essentially uses one application of f_n (costing two multiplications) and some hashing. Furthermore, at most t_1 values s_i need to be stored when such an incremental evaluation of *Verify* is used.

5 Security proof

We now formally establish the security guarantees offered by CSAR by comparing it to the ideal functionality presented in Section 3.

We first phrase the protocol in terms of an I/O machine that can be meaningfully compared to the ideal functionality in the simulatable security models. To facilitate the modeling, we include both the generation of the randomness and the verification of the proofs using *Verify* in a single machine M_P for every entity P. In a real implementation, these two algorithms would of course run on different machines; in particular, *Verify* would be evaluated several times.

Definition 4 (Real machine) *The real* $\langle$*honest*$\rangle$ [*dishonest*] *machine* M_P *for entity* P *performs the following steps:*

- *In the first activation by the environment,* $\langle M_P$ *generates the values* $(n, q_1, \ldots, q_{t_2})$ *honestly according to the randomness generation protocol*$\rangle$ [*asks the adversary for some values* $(n, q_1, \ldots, q_{t_2})$]. *This tuple* $(n, q_1, \ldots, q_{t_2})$ *is returned to the environment.*
- *In* M_P*'s second environment activation,* M_P *chooses a random* $s \in \{0,1\}^{l_1}$ *and returns* s *to the environment. The value* s *is also given to the adversary.*[4]
- *In each subsequent environment activation (the* i*-th randomness query, starting with* $i = 1$*),* $\langle M_P$ *generates the values* r_i, s_i *according to the randomness generation protocol*$\rangle$ [M_P *asks the adversary for values* r_i, s_i]. *Then* $b_i := Verify(P, n, s, r_i, q_1, \ldots, q_{t_2}, s_1, \ldots, s_i)$ *is computed.*[5] M_P *returns the triple* (r_i, s_i, b_i) *to the environment.*

The security property of CSAR can now be formally stated as follows:

Theorem 1 *Let* $l_1, l_2, t_1, t_2, t_3, \#\Pi$ *be polynomially bounded in some security parameter* k, *and* $l_2, t_2, (t_3 l_1 - l_2) \in \Omega(k)$, *and assume that the 3RSA assumption holds.*

Let a set Π *of entities be given of which an arbitrary number may be malicious. Then for any polynomial-time machine* A *there exists a polynomial-time machine* S *such that for any environment* Z *that does not access the random oracle* H *the following holds: let* P_R *denote the probability that* Z *outputs* 1 *after running together with* A *and real machines* M_P *for all* $P \in \Pi$. *Let* P_I *denote the probability that* Z *outputs* 1 *after running together with* S *and ideal machines* $\tilde{M}_P$ *for all* $P \in \Pi$. *Then* $|P_R - P_I|$ *is negligible in the security parameter* k.

Constraining the environment Z to not access the random oracle H translates into the requirement that the protocol we wish to make accountable using CSAR is not allowed to use the hash function H. This does not imply, however, that H has to be secret, since we allow the adversary to access H. (The formal consequence of disallowing Z's access to H is that the simulator now can simulate any values $H(x)$ as long as these values look random. This is crucial for our simulation proof.)

For reasons of space, we only briefly sketch the proof of Theorem 1. The full proof as well as concrete security bounds are given in Appendix A.

Proof sketch. The proof is conducted in three main steps. First, we define a variant of the real execution where the random oracle H is replaced by a simulation $\tilde{H}$. Internally, the simulation $\tilde{H}$ vastly differs from H, but it is designed to still give (almost) uniformly distributed outputs $\tilde{H}(x)$. We call the execution using $\tilde{H}$ the hybrid execution, reflecting that it is a mix of the real and the ideal execution. Then we define several events that represent various possible failures or imperfections of the simulation $\tilde{H}$, and we show that the probability $\Pr_{\text{BAD}}$ of these events is negligible. Next, we show that, unless these events occur, the outputs of $\tilde{H}$ have the same distribution as those of H. We then proceed to construct the simulator S; this construction is strongly simplified by the fact that the oracle $\tilde{H}$ already

[4] Here we simplify: Instead of using the coin-toss subprotocol, we assume that the initial seed s is chosen as true randomness. A complete treatment would have to prove that the coin-toss subprotocol presented above actually returns a truly random s. At this point, however, we treat the subprotocol as a black-box since it uses only well-known techniques.

[5] Note that the value b_i is computed correctly even for malicious P, since b_i is not part of the output of P, but represents whether or not the output of P would pass the tests.

computes all values necessary for the execution of S. Finally, we show that, unless one of the above-mentioned events occurs, the hybrid and the ideal execution have the same distribution. Hence, the distribution of the output of Z in the real and the ideal execution differ only by $\Pr_{\text{BAD}}$.

6 Implementation

We implemented CSAR as an addition to `libpeerreview`, which is an open-source implementation of PeerReview that was written by the authors of [15] and is publicly available from [25]. In total, we added or modified 1984 lines of code.

6.1 Integration with PeerReview

Our implementation is transparent to the user and works without modifications to existing application code; it simply replaces the library's `getRandom` function. When CSAR is enabled, faulty nodes can no longer predict future random values of a correct node. In addition, nodes can be exposed as faulty if they change their random seed after startup.

Internally, our code extends the application's state machine to (i) run the randomness generation protocol when a node is started for the first time, and to (ii) respond to coin-toss messages from other nodes. We could have added these elements as a meta-protocol instead, but our approach has the advantage that the additional steps can be checked natively by PeerReview. Thus, we do not need a separate mechanism to detect if a node breaks the randomness generation protocol or ignores a coin-toss message.

We also extended the log format with additional entries for the s_i. Checkpoints now include the tuple (l_2, t, i, s_i), where i is the index of the last random number generated, as well as the state of the randomness generation protocol (while it is active). This is necessary because the witnesses need to be able to start auditing from a recent checkpoint.

Our implementation uses SHA-1 hashes for H, which implies a hash length of $l_1 = 160$ bits, and it chooses the size of H^* as $t_3 = l_1 \cdot \left(\left\lceil \frac{l_2}{l_1} \right\rceil + 1\right)$. The randomness generation protocol transfers $t_2 = 5$ preimages of length $t_4 = 480$ bits. The length l_2 of the RSA modulus and the spacing t_1 between hashes in the s_i-sequence can be freely chosen by the user.

6.2 Higher efficiency with precomputation

In a straightforward implementation of CSAR, the most expensive operation is generating a random number. Verification is efficient because it only involves applying f_n to each value, and, since f_n has been chosen as $f_n(x) = x^3 \bmod n$, it can be computed with two multiplications modulo n. On the other hand, generating a random number requires evaluating $f_n^{-1}(x) = x^d \bmod n$, which involves an exponentiation modulo n and is therefore expensive.

However, we can amortize the cost of the exponentiation across several random values. We exploit that for any m and any $j \in \{1, \ldots, t_1\}$, we have that $s_{mt_1+j} = f_n^{-j}(g_m)$, where $g_m := H^*(P, mt_1, s_{mt_1})$. In particular, $s_{(m+1)t_1} = f_n^{-t_1}(g_m)$ and $s_{mt_1+j} = f_n(s_{mt_1+j+1})$ for $j = \{1, \ldots, t_1 - 1\}$. Hence, we can efficiently compute an entire block of values $s_{mt_1+1}, \ldots, s_{(m+1)t_1}$ by computing the last value first, and then deriving the other values by applying f_n $t_1 - 1$ times (this corresponds to the dashed lines in Figure 1). Additionally, note that $f_n^{-t_1}(x) \equiv x^{d^{t_1}} \equiv x^c \bmod n$ with $c := d^{t_1} \bmod \varphi(n)$. Since c needs to be computed only once, the cost for evaluating $f_n^{-t_1}$ is essentially one exponentiation modulo n.

In summary, our implementation computes the sequence s_i in blocks of t_1 values. If t_1 is sufficiently large, the amortized cost per random value is essentially two multiplications modulo n. This is confirmed by our benchmarks in Section 8.1.

7 Applications

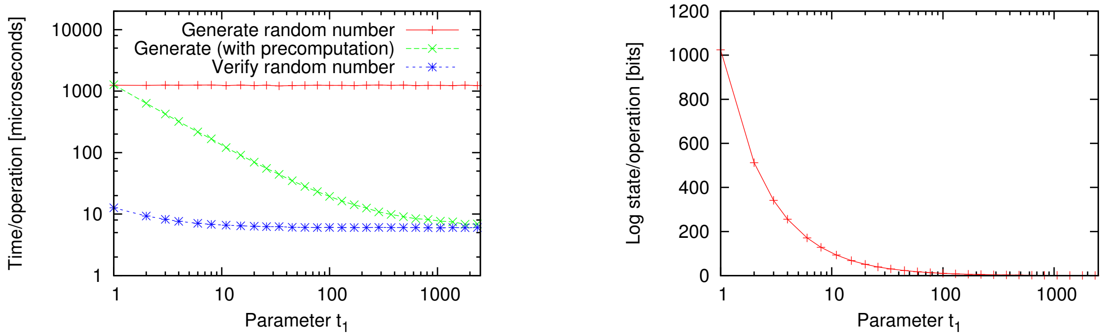
Randomness is an important instrument in the design of many distributed algorithms. Ensuring accountable pseudo-randomness is important in systems where (i) it is important to be able to detect when a node deviates from an expected sequence of pseudo-random values; and, (ii) predicting future values in a node's pseudo-random sequence may allow an attacker to gain an advantage.

In this section, we give a few examples of existing and prospective applications that use randomness in this way. In each case, CSAR can be used to add accountability to these applications without exposing them to attacks.

7.1 Sampling

Some applications use statistical sampling to estimate the properties of a large system. For example, Massoulié et al. propose a technique to aggregate statistics of peers in a peer-to-peer system using random walks or random samples [21]. A node that performs these samples must follow a pseudo-random sequence, else it could bias the results. However, if an attacker can predict future pseudo-random values generated by benign nodes, it can bias the random walk towards nodes under its own control or adjust its response to the sampling query and thereby influence the sampled value.

Random sampling is also used to measure resource usage. For example, many routers implement NetFlow [12], which provides IP flow information that ISPs use for



(a) Average time required to generate and verify a random number

(b) Average amount of state that must be revealed to the auditor per random number

Figure 2: Microbenchmarks. With $t_1 = 100$ and an RSA modulus of $l_2 = 1024$ bits, a node can generate a random number in $19\mu s$, and an auditor can verify its choice in $6.1\mu s$, given 10.2 bits of information.

billing purposes. In this case, customers wish to verify that the sampling is truly random; however, if customers were able to predict the sampling pattern, they could delay their own traffic when the ISP is about to take a sample, and thus make their resource usage appear lower.

7.2 Randomized replication

LOCKSS [20] is a distributed storage system for long-term data preservation. In LOCKSS, documents are replicated across a large number of independent storage nodes. To repair damage from data corruption, the storage nodes periodically compare their own version of each document with a number of other nodes. If there is another version that is much more common, they replace their local version with it. Many steps of this protocol are heavily randomized, so as to make it difficult for an attacker to predict the actions of a correct node.

LOCKSS would benefit from accountability because it could detect and remove faulty nodes early. However, existing techniques cannot be used because the logs would have to contain the random seeds, and thus correct nodes would be predictable. This would undermine the security of the entire system. This is not the case with CSAR, since the logs do not reveal information about a node's future actions.

7.3 Load balancing

Some systems use randomness to distribute the load evenly across a set of servers. For example, the TotalRecall storage system places replicas of objects on a random set of nodes [7]. If a node was able to predict this choice, it could insert a small dummy object whenever it knows that it will be chosen next. Thus, it could reduce its own storage load at the expense of other nodes.

A similar challenge occurs in anycast services such as [10], where requests are forwarded along a tree. If a leaf node can predict from the seed values of the interior nodes that the next request will be forwarded to it, it can insert a particularly cheap request and thus cause the more expensive requests to be forwarded to other nodes, in order to shed load unfairly.

8 Evaluation

8.1 Microbenchmarks

We begin by discussing the cost of the two fundamental operations in CSAR, namely (i) generating a random number on a node, and (ii) verifying a random number that was generated on another node. To quantify the average cost per operation, we executed each operation $10,000$ times in a tight loop, using a RSA modulus of $l_2 = 1024$ bits and varying the batching parameter t_1. The hardware we used was a Sun V20Z rack server, which has a 2.5 GHz AMD Opteron CPU. Figure 2(a) shows our results.

Without precomputation, it takes $1200\mu s$ to generate a random number, and $12.7\mu s$ to verify one. The numbers vary little with t_1, which is expected because the cost of exponentiation dominates the cost of hashing. However, if we compute random numbers in blocks of t_1 values, as described in Section 6.2, the average cost drops quickly with t_1. With $t_1 = 500$, a random number can be generated in only $9.1\mu s$ and verified in only $6.0\mu s$. This shows that our optimization is effective, and it demonstrates that the overhead from random number generation should be insignificant for most applications.

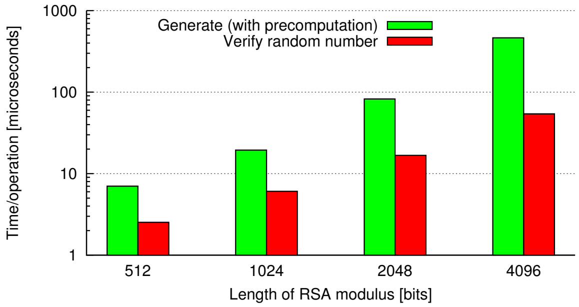


Figure 3: Key length. The cost per operation increases with the length of the RSA modulus.

In Figure 2(b), we show the average amount of state that a node must disclose to an auditor for each random value it generates. If random numbers are generated regularly, the node only needs to disclose one s_i, i.e. l_2 bits, for each block of t_1 random numbers; hence, the overhead drops quickly with t_1. With $t_1 = 500$, only 2 bits need to be disclosed on average, although one additional s_i must be disclosed during each audit if $t_1 \nmid i$. This overhead is insignificant, given that the logs of accountable applications can grow by several megabytes per hour [15].

Figure 3 shows how the average cost per operation increases with the length of the RSA modulus. For this experiment, we chose $t_1 = 100$ and used the same hardware as above.

8.2 Application-level benchmark

To estimate the overall impact of these costs, we implemented a simple demo application, which consists of a web server and k clients. The web server allows its clients to store, retrieve, or delete objects in its store, and it charges them using a simple random sampling technique: at random intervals, it picks a random file from its store, and it charges the owner one credit point. It is clearly desirable to make such a server accountable to its clients, since otherwise it might charge arbitrary amounts; however, without CSAR, this is difficult to accomplish because clients would gain the ability to predict when one of their files will be sampled, and could avoid the charge by temporarily removing that file.

We performed a simulation experiment in which we ran this server with $k = 5$ clients for one hour. On average, the server stored 1000 files with an average size of 10kB, one of which was requested every second. The expected number of samples per second was five, i.e. random numbers were used at the rather high rate of ten per second. The parameters we chose were $l_2 = 1024$ and $t_1 = 100$. We ran the simulation twice, once using CSAR to generate the random numbers and once using the `lrand` function from GLIBC (which reveals the random seed to the auditor). The workload in the two simulations was identical.

We found that CSAR changed the server's on-disk log size from 56.5 MB to 56.7 MB, a 0.3% increase. The amount of information transmitted to the auditors (the five clients) changed from 12.5 MB to 13.1 MB, a 4.2% increase. The difference occurs because the on-disk log contains additional information (such as checkpoints) which is not normally sent to the auditors. These overheads are small both in relative and absolute terms, which suggests that CSAR is practical.

9 Variants of our approach

In designing CSAR, we have made some non-obvious design choices. To highlight the importance of these choices, we now describe some possible variations of CSAR, and we point out the challenges that would have to be overcome to make them work.

9.1 Different choice of the trapdoor permutation

The most obvious variation is to use a different trapdoor one-way permutation. Although this is possible, there are a few caveats. First, our optimization technique from Section 6.2 is specific to 3RSA. Implementations using alternative permutations hence are likely to be much less efficient. Furthermore, if one replaces 3RSA by another function f, the security of CSAR will only be guaranteed if f, in addition to being one-way, satisfies the following three properties (which are derived from the security proof). First, one must be able to efficiently prove that f is indeed a permutation (this is done in CSAR by sending the values q_μ). Second, one must be able to efficiently convert a random bitstring h into an element of the domain of f (we did this by computing $v \bmod n$). Also, it must be efficiently possible to recognize if a given value is indeed in the domain of f (we did this by checking whether $s_i \in \{0, \ldots, n-1\}$). The importance of the last point is best illustrated by an example. Consider the function $f_n := x^2 \bmod n$. If n is a so-called Blum integer, then f_n is a permutation on the quadratic residues modulo n (see, e.g., [14, App. A.2.4]). However, for any given quadratic residue s_i there always exist $s_{i+1} \neq s'_{i+1}$ with $f_n(s_{i+1}) = f_n(s'_{i+1}) = s_i$ where s'_{i+1} is *not* a quadratic residue. This does not contradict the property that f_n is a permutation on the quadratic residues, but it still breaks the security of CSAR: in each step a malicious node can choose between two values, and since no efficient way is known to tell quadratic residues from quadratic non-residues, the auditors could not detect an incorrect choice.

9.2 Applying a PRG to r_i

In highly randomness-consuming protocols, one might be tempted to perform the following optimization: one generates a new r_i only when the previous r_i has been revealed (e.g., since it was contained in an audit log). Then the randomness $x_1^{(i)}, x_2^{(i)}, \ldots$ of the protocol is generated with a classical pseudo-random generator from r_i. In this case, however, a malicious node can mount the following attack: before performing some action that requires randomness, the node first checks what the next value $x_j^{(i)}$ would be. If the node does not like this value, the node delays that action until the next audit. After that audit, a new seed r_{i+1} is used and the next value is $x_1^{(i+1)}$, which possibly suits the node better. Although the effect of this attack may be small when audits are not too frequent, the possibility of such an attack is still present. Such an attack may have important consequences in protocols in which a single random value is critical, e.g., if the value determines whether a given sum of money will be transferred or not.

9.3 Using interaction

One of the limitations of CSAR is that malicious nodes can predict their own randomness. If the randomness is generated non-interactively, this is necessarily the case, since a node can always compute that randomness ahead of time. One way to circumvent this problem would be to use interactivity: for *each* random value, P performs a coin-toss with the entities in $\omega(P)$ (in this case one could also get rid of the random oracle). Although a coin-toss is a rather efficient protocol, it obviously incurs large communication costs (but this might still be feasible for protocols that only rarely need randomness). Another solution is to include the incoming messages in the generation of the randomness, i.e., $r_i := H(P, i, s_i, m)$ where m is the history of communication. Then even a malicious node can only predict its own randomness as far as it can predict incoming communication. However, this approach is flawed: if two malicious nodes collude, they can mutually influence their randomness by adaptively choosing the messages they exchange.

9.4 Using zero-knowledge

The second limitation of CSAR (which is already present in the original PeerReview) is that the auditors learn the state of a node. One can solve this problem by letting a node send only a hash of its log and then prove that the hash contains a valid log using a zero-knowledge proof. Although this is possible in theory, general purpose zero-knowledge proofs are extremely inefficient. Even the most efficient zero-knowledge proofs either target very specific number theoretic problems or need to perform a proof step for each elementary computation step in the protocol. Hence the incurred computational and communication costs would be prohibitive for all but very specific applications.

10 Conclusion

In this paper, we have described CSAR, a technique that lends accountability to systems that use randomized protocols. The key contribution is a new technique for generating cryptographically strong, accountable randomness, that is, a pseudo-random sequence that comes with a proof that the elements of the sequence have been correctly generated, while ensuring that the auditors are unable to learn anything that would make the node's future actions predictable. We have applied CSAR to a simple web server that uses random sampling for billing purposes. Our experiments indicate that the computational cost of CSAR is low and that the approach is practical: on current hardware and with a 1024-bit RSA modulus, a random number can be generated in less than $20\mu s$ and verified in less than $10\mu s$. We have additionally shown that the CSAR's storage and bandwidth costs are low both in relative and in absolute terms.

Acknowledgments

We thank the anonymous reviewers for their helpful comments.

References

[1] M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In *European Symposium on Research in Computer Security (ESORICS)*, volume 4189 of *Lecture Notes in Computer Science*. Springer, 2006.

[2] M. Backes, B. Pfitzmann, and M. Waidner. A general composition theorem for secure reactive systems. In *Proc. 1st Theory of Cryptography Conference (TCC)*, volume 2951 of *Lecture Notes in Computer Science*, pages 336–354. Springer, 2004.

[3] M. Backes, B. Pfitzmann, and M. Waidner. Secure asynchronous reactive systems. IACR Cryptology ePrint Archive 2004/082, Mar. 2004. To appear in *Information and Computation*.

[4] N. Baric and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. *Advances in Cryptology – EUROCRYPT*, pages 480–94, 1997.

[5] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security, Proceedings of CCS 1993*, pages 62–73. ACM Press,

1993. Full version online available at `http://www.cs.ucsd.edu/users/mihir/papers/ro.ps`.

[6] M. Bellare and P. Rogaway. Optimal asymmetric encryption—how to encrypt with RSA. In A. de Santis, editor, *Advances in Cryptology, Proceedings of EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1995. Extended version online available at `http://www.cs.ucsd.edu/users/mihir/papers/oae.ps`.

[7] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. TotalRecall: System support for automated availability management. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, Mar 2004.

[8] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001. Extended version in Cryptology ePrint Archive, Report 2000/67, `http://eprint.iacr.org/`.

[9] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. In *Thirtieth Annual ACM Symposium on Theory of Computing, Proceedings of STOC 1998*, pages 209–218. ACM Press, 1998. Preliminary version, extended version online available at `http://eprint.iacr.org/1998/011.ps`.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable application-level anycast for highly dynamic groups. In *NGC 2003*, Sep 2003.

[11] M. Chase and A. Lysyanskaya. Simulatable VRFs with applications to multi-theorem NIZK. In A. Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 303–322. Springer, 2007.

[12] B. Claise. RFC 3954: Cisco systems NetFlow services export version 9. `http://www.ietf.org/rfc/rfc3954.txt`, Oct 2004.

[13] R. Dingledine, M. J. Freedman, and D. Molnar. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Accountability. O'Reilly and Associates, 2001.

[14] O. Goldreich. *Foundations of Cryptography – Volume 1 (Basic Tools)*. Cambridge University Press, Aug. 2001. Previous version online available at `http://www.wisdom.weizmann.ac.il/~oded/frag.html`.

[15] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 175–188. ACM, Oct 2007.

[16] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[17] D. Knuth. *The art of computer programming. 2. Seminumerical algorithms*. Addison-Wesley, 2 edition, 1969.

[18] L. Lamport. Password authentication with insecure communication. *Commun. ACM*, 24(11):770–772, 1981.

[19] B. W. Lampson. Computer security in the real world. In *Proc. Annual Computer Security Applications Conference*, Dec 2000.

[20] P. Maniatis, D. S. H. Rosenthal, M. Roussopoulos, M. Baker, T. Giuli, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 44–59. ACM, 2003.

[21] L. Massoulié, E. L. Merrer, A.-M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing (PODC'06)*, pages 123–132. ACM, 2006.

[22] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science*, pages 120–130, New York, NY, October 1999. IEEE.

[23] N. Michalakis, R. Soulé, and R. Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, Apr 2007.

[24] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Mar 2003.

[25] PeerReview project homepage. `http://peerreview.mpi-sws.org/`.

[26] RSA Laboratories. *PKCS #1: RSA Cryptography Standard, Version 2.1*, 2002. Online available at `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf`.

[27] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3):281–292, 1971.

[28] A. R. Yumerefendi and J. S. Chase. Trust but verify: Accountability for Internet services. In *ACM SIGOPS European Workshop*, Sep 2004.

[29] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the First Workshop on Hot Topics in System Dependability (HotDep'05)*, Jun 2005.

[30] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, Feb 2007.

A Security Proof

Theorem 2 *Let a set Π of nodes be given of which an arbitrary number may be malicious. Then for any polynomial time machine A there exists a polynomial time machine S such that for any environment Z that does not access the random oracle H the following holds: Let P_R denote the probability that Z outputs 1 after running together with A and real machines M_P for all $P \in \Pi$. Let P_I denote the probability that Z outputs 1 after running together with S*

and ideal machines $\tilde{M}_P$ for all $P \in \Pi$. Then

$$|P_R - P_I| \\ \leq (\tfrac{3}{7} + \tfrac{4}{7} \cdot 2^{l_2 - t_3 l_1})^{t_2} \cdot Q + (2^{-l_2+1} + 2^{l_2 - t_3 l_1}) \cdot Q t_1 \\ + 2^{l_2 - t_3 l_1} \cdot Q^2 \cdot \#\Pi \\ + Q \cdot \#\Pi \cdot \varepsilon_{3\mathrm{RSA}}(l_2, O(T + Q t_3 l_1 + \\ (Q + \#\Pi t_2) l_2^2 \log l_2 \log\log l_2))$$

Here Q denotes the number of queries performed by Z and A (both to the randomness generation protocol and to H), T denotes an upper bound on the size of the circuits describing Z and A (i.e., roughly the running time) and $\#\Pi$ is the number of nodes.

In particular, if $l_1, l_2, t_1, t_2, t_3, \#\Pi$ are polynomially bounded in some security parameter k, and $l_2, t_2, (t_3 l_1 - l_2) \in \Omega(k)$, and the 3RSA assumption holds, then $|P_R - P_I|$ is negligible.

A.1 Proof plan

To prove theorem 2, we proceed in three main steps. First, we define a variant of the real execution where the random oracle H is replaced by a simulation $\tilde{H}$ which internally works very differently from H but is designed to still give (almost) uniformly distributed outputs $\tilde{H}(x)$. We call the execution using $\tilde{H}$ the hybrid execution (since it is a mix between the real and the ideal execution). Then several events are defined that represent various possible failures or imperfections of the simulation $\tilde{H}$ and the probability $\Pr_{\mathrm{BAD}}$ of these events is then shown to be negligible. It is then shown that unless these events occur, the outputs of $\tilde{H}$ have the same distribution as those of H. We then proceed to construct the simulator S which is strongly simplified by the fact that the oracle $\tilde{H}$ already computes all values necessary for the execution of S. We then show that unless one of the above-mentioned events occurs, the hybrid and the ideal execution have the same distribution. Concluding, we have that the distribution of the output of Z in the real and the ideal execution differ only by $\Pr_{\mathrm{BAD}}$.

Before we present the actual proof, we need the following auxiliary lemma:

Lemma 1 *Let $n \in \mathbb{N}$ be an integer of length l_2. Let $l_y \geq l_2$. If $f_n(x) := x^3 \bmod n$ is not a permutation on $\{0, \dots, n-1\}$, then for random $y \in \{0,1\}^{l_y}$ the probability that some value q exists with $f_n(q) \equiv y \bmod n$ is bounded from above by $\frac{3}{7} + \frac{4}{7} \cdot 2^{l_2 - l_y}$.*

Proof: For $n \leq 2$ the function f_n is always a permutation. Thus assume $n \geq 3$. Since f_n is a permutation iff $3 \nmid \varphi(n)$ (since exactly in this case 3 has a multiplicative inverse modulo $\varphi(n)$) we know that $3 \mid \varphi(n)$. Thus there is a prime p and an $e \in \mathbb{N}$ such that $p^e \mid n$, $p^{e+1} \nmid n$ and $3 \mid \varphi(p^e)$. We distinguish two cases, $p = 3$ and $p \neq 3$. If $p = 3$, we have that $\varphi(p^e) = 2 \cdot 3^{e-1}$, thus $e > 2$ and $9 \mid n$. Thus if $q^3 \equiv y \bmod n$, then $q^3 \equiv y \bmod 9$. The only solutions to this equation are $q \in \{0, 1, 8\} \bmod 9$. Thus, for random $y \bmod n$ (and thus also random $y \bmod 9$), we have that there exists a q with $q^3 \equiv y \bmod n$ with probability $\frac{1}{3} \leq \frac{3}{7}$. Now we consider the case $p \neq 3$. Since $\varphi(2^e) = 2^{e-1}$ and $\varphi(5^e) = 4 \cdot 5^{e-1}$, by $3 \mid \varphi(p^e)$ we have $p \geq 7$. Further, since $3 \nmid p$ and $3 \mid \varphi(p^e) = (p-1)p^{e-1}$, we have that $3 \mid p - 1$. The operation $f_p : x \mapsto x^3 \bmod p$ corresponds to the function $\tilde{f}_p : \tilde{x} \mapsto 3\tilde{x} \bmod p - 1$ (because $\mathbb{Z}_p^\times \cong \mathbb{Z}_{\varphi(p)} = \mathbb{Z}_{p-1}$ where $\mathbb{Z}_p^\times$ is the multiplicative group of $\mathbb{Z}_p$). Since $\tilde{f}_p(\mathbb{Z}_{p-1}) \cong \mathbb{Z}_{\frac{p-1}{3}}$, the number of $\tilde{y} \in \mathbb{Z}_{p-1}$ that have a preimage under $\tilde{f}_p$ is at most $\frac{p-1}{3}$. Thus the number of $y \in \mathbb{Z}_p^\times$ with a preimage under f_p is also at most $\frac{p-1}{3}$, and the number of $y \in \mathbb{Z}_p$ with a preimage under f_p is at most $N := \frac{p-1}{3} + 1$ (since $\mathbb{Z}_p \setminus \mathbb{Z}_p^\times = \{0\}$). Thus for random $y \bmod p$ a preimage under f_p exists with probability at most $\frac{N}{p}$, and thus a random $y \bmod n$ has a preimage under f_n with probability at most $\frac{N}{p}$. Since $p \geq 7$ we have $\frac{N}{p} = \frac{p+2}{3p} \leq \frac{7+2}{3 \cdot 7} = \frac{3}{7}$. So altogether, when choosing y such that $y \bmod n$ is uniformly distributed on $\{0, \dots, n-1\}$, the probability that $y \bmod n$ has a preimage under f_n is at most $\frac{3}{7}$. Fix u, v with $un + v = 2^{l_y}$ and $v \in \{0, \dots, n-1\}$. Since $2^{l_y} \geq n$ we have $u \geq 1$. Let y be randomly chosen from $\{0,1\}^{l_y}$. Then with probability $P := \frac{un}{un+v} = 1 - \frac{v}{un+v} \geq 1 - 2^{l_2 - l_y}$ we have that $y \in \{0, \dots, un\} := M$. Under the condition that $y \in M$ we have that $y \bmod n$ is uniformly distributed on $\{0, \dots, n-1\}$. Thus the probability that $y \bmod n$ has a preimage under f_n is bounded by $\frac{3}{7} \cdot P + (1 - P) \leq \frac{3}{7} \cdot (1 - 2^{l_2 - l_y}) + 2^{l_2 - l_y} = \frac{3}{7} + \frac{4}{7} \cdot 2^{l_2 - l_y}$. $\square$

A.2 Simulating the random oracle

Here and for the rest of the proof, for an integer i, let m and j always be integers such that $mt_1 + j = i$ and $j \in \{1, \dots, t_1\}$.

In the first step, we replace the random oracle H in the real execution by a lazily sampled function $\tilde{H}$. The real execution with $\tilde{H}$ we call the *hybrid execution*. The oracle $\tilde{H}$ acts as follows:

1. First, for each node P an infinite sequence of random values $r_i^P \in \{0,1\}^{l_1}$ is randomly chosen. Further, $g_m^P := \bot$ for all $m \in \mathbb{N}$ and all nodes P. Initially set $\tilde{H}(x) := \bot$ for all x. Let $G := \varnothing$ and $N := \varnothing$. When we say "sample $\tilde{H}(x)$" we mean "if $\tilde{H}(x) = \bot$, choose a random $h \in \{0,1\}^{l_1}$ and set $\tilde{H}(x) := h$".
2. Let s^P denote the value s chosen by node P. Until s^P has been chosen, let $s^P := \bot$. Let n^P denote the public key n chosen by node P and let $q_1^P, \dots, q_n^P$ be

the values $q_1, \dots, q_{t_2}$ output by P. As soon as n^P has been output, sample $\tilde{H}(\mathtt{pk}, \mu, n^P)$ for $\mu = 1, \dots, t_2$. Check whether $f_{n^P}(q_\mu) = \tilde{H}(\mathtt{pk}, \mu, n^P)$ for all $\mu = 1, \dots, t_2$. If so, set $N := N \cup \{n^P\}$.

3. As soon as s^P is determined, sample $\tilde{H}(P, \mathtt{start}, s^P)$, and set $s_0^P := \tilde{H}(P, \mathtt{start}, s^P)$. Then sample $\tilde{H}(\mu, P, 0, s_0^P)$ for $\mu = 1, \dots, t_2$. Then set $g_0^P := \tilde{H}^*(P, 0, s_0^P)$ and $G := G \cup \{(P, 0)\}$.
4. Upon a query $\tilde{H}(x)$ do the following:
    - Check whether $x = (P, i, \tilde{x})$ or $x = (\mu, P, i, \tilde{x})$ such that the following holds: (i) i is an integer and $i \geq 1$, (ii) P is a node, (iii) $n^P \in N$, (iv) $(P, m) \in G$, (v) $\tilde{x} \in \{0, \dots, n^P - 1\}$, (vi) $f_{n^P}^j(\tilde{x}) \equiv g_m^P \bmod n^P$,
    - If this check succeeds, set $\tilde{H}(P, i, \tilde{x}) := r_i^P$. Further, if additionally $j = t_1$, sample $\tilde{H}(\mu, P, i, \tilde{x})$ for $\mu = 1, \dots, t_2$ and set $g_{m+1}^P := \tilde{H}^*(P, i, \tilde{x})$ and $G := G \cup \{(P, m)\}$.
    - Finally, sample $\tilde{H}(x)$ and return $\tilde{H}(x)$.[6]

A.3 Events

We define the following events that may occur in the hybrid execution:

Event GCONFLICT: Queries $\tilde{H}(x_1)$, $\tilde{H}(x_2)$ are performed with $x_1 = (P, i, \tilde{x})$ or $x_1 = (\mu, P, i, \tilde{x})$ and with $x_2 = (P, i, \tilde{x})$ or $x_2 = (\mu', P, i, \tilde{x})$ (note that P, i, and $\tilde{x}$ are the same in both queries) such that during the first query we have $(P, m) \notin G$ and during the second query we have $(P, m) \in G$, $n^P \in N$ and $f_{n^P}^j(\tilde{x}) \equiv g_m^P \bmod n^P$.

Event REASSIGN: $\tilde{H}(x)$ is assigned a value y although it was already assigned some value $y' \notin \{y, \perp\}$.

Event ALIAS: In two queries to $\tilde{H}$, two triples $(P, i, \tilde{x})$ and $(P, i, \tilde{x}')$ with $\tilde{x} \neq \tilde{x}'$ (but with the same P, i) pass the check in Step 4.

Event NONINJECTIVE: There exists a node P such that $n^P \in N$ and f_{n^P} is *not* a permutation on $\{0, \dots, n^P - 1\}$.

Event PREDICT: In some query to $\tilde{H}$, a triple $(P, i, \tilde{x})$ passes the check in Step 4 where P is an honest node and the i-th randomness query to P by the environment has not yet been performed. (Here, if the query to $\tilde{H}$ occurs *during* the i-th randomness query to P, we consider the i-th randomness query as already performed.)

Event WRONGPROOF: An honest party outputs a triple $(\tilde{r}_i, \pi_i, b_i)$ in a randomness query with $b_i \neq 1$.

Event WRONGRANDOM: A party outputs a triple $(\tilde{r}_i, \pi_i, b_i)$ with $\tilde{r}_i \neq r_i$ and $b_i = 1$.

Event BAD: One of the events GCONFLICT, REASSIGN, ALIAS, NONINJECTIVE, PREDICT, WRONGPROOF, or WRONGRANDOM occurs.

[6]Note that sampling $\tilde{H}(x)$ only has an effect if $\tilde{H}(x)$ has not been assigned in the preceding step.

A.4 Event probabilities

We will now bound the probabilities of the various events defined above.

First, we bound the probability of NONINJECTIVE. A value n is in N only if $\tilde{H}^*(\mathtt{pk}, \mu, n)$ has a preimage modulo n under f_n for all $\mu \in \{1, \dots, t_2\}$. Since the values $h_\mu := \tilde{H}^*(\mathtt{pk}, \mu, n)$ are uniformly chosen from $\{0,1\}^{t_3 l_1}$ with $t_3 l_1 \geq l_2 \geq |n|$ (and n cannot depend on the h_μ since n is given as argument to $\tilde{H}^*$), we have by lemma 1 that if f_n is not a permutation, the probability that all h_μ have a preimage under f_n is at most $(\frac{3}{7} + \frac{4}{7} \cdot 2^{l_2 - t_3 l_1})^{t_2}$. Thus, since at most Q different values n can be queried, the probability of NONINJECTIVE is at most $(\frac{3}{7} + \frac{4}{7} \cdot 2^{l_2 - t_3 l_1})^{t_2} Q$.

Next, we bound the probability of GCONFLICT $\wedge$ $\neg$NONINJECTIVE. Assume that GCONFLICT $\wedge$ $\neg$NONINJECTIVE occurs. Then in the first query $\tilde{H}(x_1)$, the value g_m^P has not yet been chosen. Further, in the second query we have that $n^P \in N$, thus f_{n^P} is a permutation on $\{0, \dots, n^P - 1\}$ (since we assume $\neg$NONINJECTIVE). Further, in the second query we have that $f_{n_P}^j(\tilde{x}) \equiv g_m^P \bmod n_P$ and thus $\tilde{x} \equiv f_{n_P}^{-j}(g_m^P) \bmod n_P$ where g_m^P is randomly chosen *after* $\tilde{x}$ (because $\tilde{x}$ is already used in the first query). Since g_m^P is uniformly distributed on $\{0,1\}^{t_3 l_1}$, we have that $g_m^P \bmod n^P$ is uniformly distributed on $\{0, \dots, n^P - 1\}$ under the condition that $g_m^P < 2^{t_3 l_1} - (2^{t_3 l_1} \bmod n)$. The probability that $g_m^P \geq 2^{t_3 l_1} - (2^{t_3 l_1} \bmod n)$ is at most $\frac{n}{2^{t_3 l_1}} \leq 2^{l_2 - t_3 l_1}$, thus the statistical distance δ between the distribution of $g_m^P \bmod n^P$ and the uniform distribution on $\{0, \dots, n^P - 1\}$ is at most $2^{l_2 - t_3 l_1}$. Since $f_{n^P}^{-j}$ is a permutation, the same holds for $f_{n^P}^{-j}(g_m^P) \bmod n^P$. Thus the probability that a random g_m^P fulfills $\tilde{x} \equiv f_{n^P}^j(g_m^P)$ is at most $\frac{1}{n} + \delta \leq 2^{-l_2+1} + 2^{l_2 - t_3 l_1}$. Since at most Q different queries $\tilde{H}(x_1)$ can be performed in an execution, and j can take only t_1 different values (we have $j \in \{1, \dots, t_1\}$), we have that the probability that GCONFLICT $\wedge$ $\neg$NONINJECTIVE occurs is at most $(2^{-l_2+1} + 2^{l_2 - t_3 l_1}) Q t_1$.

Now, we show that REASSIGN $\wedge$ $\neg$GCONFLICT does not occur. By our definition of sampling, sampling $\tilde{H}(x)$ for some x can never reassign $\tilde{H}(x)$. Thus the only place where some $\tilde{H}(x)$ could be reassigned is in Step 4, namely the assignment $\tilde{H}(P, i, \tilde{x}) := r_i^P$. However, this assignment can only occur if $(P, m) \in G$, $n^P \in N$ and $f_{n^P}^j(\tilde{x}) \equiv g_m^P \bmod n^P$. Further, for this assignment to be a reassignment, $\tilde{H}(P, i, \tilde{x})$ needs to have already been assigned a different value, i.e., $\tilde{H}(P, i, \tilde{x})$ needs to have been sampled in an earlier query. For this, in the earlier query $(P, m) \notin G$ needs to hold (otherwise the check in Step 4 would have been passed). Thus REASSIGN implies GCONFLICT, and therefore REASSIGN $\wedge$ $\neg$GCONFLICT does not occur.

Now we show that ALIAS $\wedge\neg$NONINJECTIVE never occurs. ALIAS occurs if two triples $(P, i, \tilde{x})$ and $(P, i, \tilde{x}')$ with $\tilde{x} \neq \tilde{x}'$ pass the test in Step 4. This implies that $n^P \in N$, that $\tilde{x}, \tilde{x}' \in \{0, \dots, n^P - 1\}$, and that $f^j_{n^P}(\tilde{x}) = g^P_m = f^j_{n^P}(\tilde{x}')$. This is only possible if $f^j_{n^P}$ is not a permutation on $\{0, \dots, n^P - 1\}$. However, this would imply NONINJECTIVE since $n^P \in N$. Thus ALIAS $\wedge \neg$NONINJECTIVE never occurs.

We now show that event WRONGRANDOM $\wedge$ $\neg$REASSIGN never occurs. Both honest and malicious machines M_P set $b_i := \mathit{Verify}(P, n^P, s^P, \tilde{r}^P_i, q^P_1, \dots, q^P_{t_2}, s^P_1, \dots, s^P_i)$ where $(\tilde{r}^P_i, s^P_i)$ are the values chosen by the adversary in the i-th randomness query to P. Assume that no $\tilde{H}(x)$ is ever reassigned a different value, i.e., that REASSIGN does not occur. A comparison of the definition of *Verify* and Step 4 of the simulation of $\tilde{H}$ then reveals that if *Verify* returns $b_i = 1$, then the simulation of $\tilde{H}$ sets $\tilde{H}(P, i, s^P_i)$ to r^P_i. Since *Verify* only returns $b_i = 1$ if $\tilde{r}^P_i = \tilde{H}(P, i, s^P_i)$, it follows that if $b_i = 1$ then $\tilde{r}^P_i = r^P_i$. Thus WRONGRANDOM $\wedge\neg$REASSIGN never occurs.

We now show that WRONGPROOF $\wedge\neg$REASSIGN does not occur. By construction of the protocol, as long as the oracle $\tilde{H}$ always returns the same value on the same input (i.e., REASSIGN does not occur), all checks in the definition of *Verify* succeed, thus WRONGPROOF $\wedge$ $\neg$REASSIGN does not occur.

A.5 Bounding the probability of PREDICT

Now we bound the probability of PREDICT $\wedge\neg$REASSIGN. This is actually the only place in this proof where the one-wayness of f_n comes into play. Let γ be the probability that PREDICT $\wedge\neg$REASSIGN occurs. Then, for a random honest node $\hat{P}$ and a random integer $\hat{\imath} \in \{1, \dots, Q\}$, the probability that PREDICT $\wedge\neg$REASSIGN occurs with $P = \hat{P}$ and $i = \hat{\imath}$ is at least $\frac{\gamma}{Q\#\Pi}$. We can then transform the whole system consisting of nodes, environment, adversary, and $\tilde{H}$ into one machine *Sim* that performs the following:

- First, it chooses a random RSA modulus $\hat{n}$ of length l_2 with $3 \nmid \varphi(n)$ and a random $\hat{y} \in \{0, \dots, n-1\}$.
- Then it chooses a random honest node $\hat{P}$ and an integer $\hat{\imath} \in \{1, \dots, Q\}$. It computes $\hat{m}$ and $\hat{\jmath} \in \{1, \dots, t_1\}$ such that $\hat{\imath} = \hat{m}t_1 + \hat{\jmath}$.
- It simulates the hybrid execution with the following modifications:
  - (i) When $\hat{P}$ would choose the RSA modulus $n^{\hat{P}}$, it sets instead $n^{\hat{P}} := \hat{n}$.
  - (ii) When $H^*(\mathtt{pk}, \mu, \tilde{x})$ is to be sampled,[7] choose some random $q \in \{0, \dots, \hat{n} - 1\}$ and choose a random $\hat{h}_\mu \in \{0,1\}^{t_3 l_1}$ with $\hat{h}_\mu \equiv f_{\hat{n}}(q) \bmod \hat{n}$. Store $(q, \hat{h}_\mu)$ in some list L.
  - (iii) When in Step 4 of the simulation of $\tilde{H}$, the value $\tilde{H}^*(\hat{P}, i, \tilde{x})$ is to be sampled, do not choose these values randomly but choose a random $\hat{g}_m \in \{0,1\}^{t_3 l_1}$ with $\hat{g}_m \equiv g \bmod \hat{n}$ where g is chosen as follows: If $m = \hat{m}$, then $g := f^{\hat{\jmath}-1}_{\hat{n}}(\hat{y})$, and if $m \neq \hat{m}$, choose a random $y' \in \{0, \dots, n-1\}$ and set $g := f^{t_1}_{\hat{n}}(y')$. In this computation, on each invocation of $f_{\hat{n}}(a) = b$, store (a, b) in the list L. Then assign $\hat{g}_m$ to $\tilde{H}^*(\hat{P}, i, \tilde{x})$.
  - (iv) When $M_{\hat{P}}$ computes $f^{-1}_{\hat{n}}(b)$ for some x, search for some (a', b') with $b = b'$ in L and return a. Only if no such (a', b') exists, use the secret key corresponding to $\hat{n}$ to compute $f^{-1}_{\hat{n}}(b)$.

Note that in this simulation, $n^{\hat{P}}$ is chosen with the same distribution as in the hybrid execution. Further, the computation of $f^{-1}_{\hat{n}}$ by $M_{\hat{P}}$ is performed differently, but the result is the same as in the hybrid execution since if $(a', b') \in L$ then $b' = f_{\hat{n}}(a')$ and thus $a' = f^{-1}_{\hat{n}}(b')$ (note that since $\hat{n}$ is chosen honestly, $f_{\hat{n}}$ is a permutation). Now consider the choice of $\hat{g}_m$. These values are not chosen uniformly from $\{0,1\}^{t_3 l_1}$, but instead they are chosen uniformly under the precondition that $\hat{g}_m \equiv g \bmod \hat{n}$. The value g is chosen uniformly from $\{0, \dots, \hat{n} - 1\}$ (since $f_{\hat{n}}$ is a permutation, and y' is each time a fresh random value and $\hat{y}$ is only used for $\hat{g}_{\hat{m}}$). Thus $\hat{g}_m$ is a fresh random value with a distribution that has a statistical distance δ from the uniform distribution with $\delta \leq \frac{2^{t_3 l_1} \bmod \hat{n}}{2^{t_3 l_1}} \leq \frac{\hat{n}}{2^{t_3 l_1}} \leq 2^{l_2 - t_3 l_1}$. Analogous reasoning holds for $\hat{h}_\mu$. Since at most Q values $\hat{g}_m$ and $\hat{h}_\mu$ are chosen, the overall error introduced at most is $2^{l_2 - t_3 l_1} \cdot Q$. Thus the probability that PREDICT $\wedge$ $\neg$REASSIGN occurs in the execution simulated by *Sim* is at least $\frac{\gamma}{Q\#\Pi} - 2^{l_2 - t_3 l_1} \cdot Q$. The machine $M_{\hat{P}}$ computes $f^{-1}_{\hat{n}}$ only in two situations. First, for computing $q_\mu = f^{-1}_{\hat{n}}(\tilde{H}^*(\mathtt{pk}, \mu, \hat{n}))$ and second for computing $s_i = f^{-j}_{\hat{n}}(\tilde{H}^*(P, mt, s_{mt}))$. In the first case, after querying $h := \tilde{H}^*(\mathtt{pk}, \mu, \hat{n})$, a pair $(q, \hat{h}_\mu)$ with $\hat{h}_\mu = h$ is contained in L. Thus $f^{-1}_{\hat{n}}$ is computed without accessing the secret key. In the second case, when computing s_i, as long as REASSIGN does not occur, the value $g_m := \tilde{H}^*(P, mt, s_{mt})$ is chosen in Step (iii) as $\hat{g}_m$. In this case, for $i < \hat{\imath}$ (and thus $m < \hat{m}$ or $j < \hat{\jmath}$), we have that $(f^{-j}_{\hat{n}}(\hat{g}_m), f^{-j+1}_{\hat{n}}(\hat{g}_m)) \in L$ and s_i is computed without accessing the secret key of $M_{\hat{P}}$. Thus $M_{\hat{P}}$ does not use its secret key before the i-th randomness query unless REASSIGN occurs. If PREDICT occurs with $P = \hat{P}$ and $i = \hat{\imath}$, we have that a triple $(\hat{P}, \hat{\imath}, \tilde{x})$ is accepted in

[7] When we say that $H^*(x)$ is to be sampled, we mean that $H(k, x)$ is to be sampled for some $k \in \{1, \dots, t_3\}$. Similarly, when assigning some value $v_1 \| \dots v_{t_3}$ to $H^*(x)$, we assign v_k to $H^*(k, x)$. We use this somewhat sloppy notation for readability.

Step 4 of the simulation of $\tilde{H}$ before the i-th randomness query of $M_{\hat{P}}$. This implies that $\tilde{x} = f_{\hat{n}}^{-\hat{j}}(\hat{g}_{\hat{m}}) = \tilde{x} = f_{\hat{n}}^{-\hat{j}}(f_{\hat{n}}^{\hat{j}-1}(\hat{g}_{\hat{m}})) = f_{\hat{n}}^{-1}(\hat{y})$. So, if PREDICT$\wedge\neg$REASSIGN, Sim finds a preimage of $\hat{y}$ under $f_{\hat{n}}$ without accessing the secret key corresponding to n if PREDICT $\wedge \neg$REASSIGN occurs with $P = \hat{P}$ and $i = \hat{\imath}$. Since the probability for this is at least $\frac{\gamma}{Q\#\Pi} - 2^{l_2 - t_3 l_1} \cdot Q$ as seen above, by definition of $\varepsilon_{3\mathrm{RSA}}$ we have that $\frac{\gamma}{Q\#\Pi} - 2^{l_2-t_3l_1} \cdot Q \le \varepsilon_{3\mathrm{RSA}}(l_2, S)$ where S is the size of the circuit describing the machine Sim. It can be easily verified Sim can be described by a circuit of size $O(T + Qt_3l_1 + (Q + \#\Pi t_2)X)$ where X is the size of a circuit that performs an exponentiation modulo a number n of length l_2. By [27] and [17, p. 295] we have $X \in O(l_2^2 \log l_2 \log\log l_2)$. Thus the probability γ that PREDICT $\wedge \neg$REASSIGN occurs is at most $Q \cdot \#\Pi \cdot \varepsilon_{3\mathrm{RSA}}(l_2, O(T+Qt_3l_1+(Q+\#\Pi t_2)l_2^2 \log l_2 \log\log l_2)) + 2^{l_2-t_3l_1} \cdot Q^2 \cdot \#\Pi$.

A.6 The probability of BAD

The event BAD is equivalent to NONINJECTIVE $\vee$ (GCONFLICT $\wedge$ $\neg$NONINJECTIVE) $\vee$ (REASSIGN $\wedge$ $\neg$GCONFLICT) $\vee$ (ALIAS $\wedge$ $\neg$NONINJECTIVE) $\vee$ (WRONGRANDOM $\wedge$ $\neg$REASSIGN) $\vee$ (WRONGPROOF $\wedge$ $\neg$REASSIGN) $\vee$ (PREDICT $\wedge$ $\neg$REASSIGN). Combining the above bounds on the probabilities of the various events, we get that BAD occurs with probability at most

$$
\begin{aligned}
&\Pr{}_{\mathrm{BAD}} \\
&:= (\tfrac{3}{7} + \tfrac{4}{7} \cdot 2^{l_2 - t_3 l_1})^{t_2} \cdot Q + (2^{-l_2+1} + 2^{l_2-t_3l_1}) \cdot Qt_1 \\
&+ Q \cdot \#\Pi \cdot \varepsilon_{3\mathrm{RSA}}(l_2, O(T + Qt_3l_1 + \\
&\qquad (Q + \#\Pi t_2)l_2^2 \log l_2 \log\log l_2)) \\
&+ 2^{l_2-t_3l_1} \cdot Q^2 \cdot \#\Pi
\end{aligned}
$$

A.7 Faithfulness of the oracle simulation

We will now show that the simulation of $\tilde{H}$ as described above is a faithful simulation of the random oracle H. More exactly, we show that unless REASSIGN or ALIAS occurs we have that when $\tilde{H}$ is queried twice with the same value it returns the same image, and when $\tilde{H}$ is queried with a value x that has not yet been queried, a fresh random value from $\{0,1\}^{l_1}$ is returned.[8] Since the simulated $\tilde{H}$ upon query x always returns $\tilde{H}(x)$ (where the partial function $\tilde{H}$ is possibly modified first), $\tilde{H}$ will always return the same values on the same queries unless REASSIGN occurs.

To see that for a value x that has not yet been queried, a fresh random value is returned, note that there are only two possibilities how $\tilde{H}(x)$ gets assigned a value. First, $\tilde{H}(x)$ is sampled. In this case, by definition $\tilde{H}(x)$ is assigned a fresh value. Or second, $\tilde{H}(P,i,\tilde{x})$ is assigned r_i^P. Since each r_i^P is an independently chosen random value, and that value is never accessed until r_i^P is assigned, $\tilde{H}(P,i,\tilde{x})$ is assigned only fresh random values unless some r_i^P is assigned to two different $\tilde{H}(P,i,\tilde{x})$. This again only happens if for two triples $(P,i,\tilde{x})$ and $(P,i,\tilde{x}')$ with $\tilde{x} \neq \tilde{x}'$ pass the check in Step 4 (in different queries), i.e., if ALIAS occurs. Thus unless REASSIGN or ALIAS occurs, $\tilde{H}$ is a faithful simulation of a random oracle.

A.8 Constructing the simulator

For a given adversary A that runs with the real machines M_P, we now construct the simulator S that runs with the ideal machines $\tilde{M}_P$ in the ideal execution. This simulator S does the following:

(i) It simulates the random oracle $\tilde{H}$ as described above. However, it does not choose the values r_i^P on its own but uses the values r_i chosen by machine $\tilde{M}_P$. By definition, malicious machines make the r_i accessible to the simulator. If $\tilde{M}_P$ is honest, and a value r_i is required that $\tilde{M}_P$ has not yet sent to the simulator, the simulator aborts.

(ii) It simulates an instance of the adversary A. Any communication from the environment to the simulator is passed to the simulated adversary A.

(iii) When an ideal honest machine $\tilde{M}_P$ requests a tuple $(n, q_1, \dots, q_{t_2})$, the simulator computes $(n, q_1, \dots, q_{t_2})$ according to the protocol (i.e., n is an RSA modulus and $q_\mu := f_n^{-1}(\tilde{H}(\mathtt{pk}, \mu, n))$).

(iv) When an ideal malicious machine $\tilde{M}_p$ requests a tuple $(n, q_1, \dots, q_{t_2})$ that request is forwarded to the adversary A.

(v) When the machine $\tilde{M}_P$ passes the value s to the simulator, that value is forwarded to the adversary.

(vi) When the malicious machine $\tilde{M}_P$ requests a triple $(\tilde{r}_i, s_i, b_i)$, the simulator requests $(\tilde{r}_i, s_i)$ from the adversary A, and computes $b_i := \mathit{Verify}(P, n, s, \tilde{r}_i, q_1, \dots, q_{t_2}, s_1, \dots, s_i)$ where n, s, q_μ, s_μ are the respective values output by $\tilde{M}_P$. Then the simulator returns $(\tilde{r}_i, s_i, b_i)$ to $\tilde{M}_P$.

(vii) When the honest machine $\tilde{M}_P$ requests a triple $(\tilde{r}_i, s_i, b_i)$, the simulator sets $b_i := 1$ and computes $(\tilde{r}_i, s_i)$ according to the honest protocol (i.e., $\tilde{r}_i := \tilde{H}(P, i, s_i^P)$ s_i is computed recursively as $f_n^{-1}(s_{i-1})$ or $f_n^{-1}(\tilde{H}^*(P, i-1, s_{i-1}))$ or $\tilde{H}^*(P, \mathtt{start}, s)$, respectively). Note that the simulator is able to compute f_n^{-1} for honest machines $\tilde{M}_P$ since the simulator has chosen the modulus n for $\tilde{M}_P$ himself.

[8] By *fresh* we mean that this value is uniformly distributed and independent of all other values returned so far.

A.9 Faithfulness of the simulation

We will now show that the view of the environment is identical in an execution of the adversary A and the real machines M_P but with simulated $\tilde{H}$ (the hybrid execution) and in an execution of the simulator S and the ideal machines $\tilde{M}_P$ (the ideal execution) unless PREDICT, WRONGRANDOM or WRONGPROOF occurs. Steps (i)–(v) are a direct simulation of the corresponding actions of the real machines and the adversary unless the simulator aborts in Step (i). The latter only happens when a value r_i^P is required that has not yet been given by the honest $\tilde{M}_P$ to the simulator, i.e., if PREDICT occurs.

Consider Step (vi). In the hybrid execution the malicious machine M_P returns the triple $(\tilde{r}_i, s_i, b_i)$ where $b_i := \mathit{Verify}(P, n, s, \tilde{r}_i, q_1, \ldots, q_{t_2}, s_1, \ldots, s_i)$ and $(\tilde{r}_i, s_i)$ are the values chosen by the adversary A. In the ideal execution, the malicious machine $\tilde{M}_P$ returns the triple (r'_i, s_i, b_i) where s_i is the value chosen by A and b_i is computed as in the hybrid execution. Further we have $r'_i = \tilde{r}_i$ if $b_i = 0$ and $r'_i = r_i$ if $b_i = 1$ (here r_i is the random value chosen by $\tilde{M}_P$ itself). Thus the triples returned in the hybrid and the ideal execution are equal unless $r_i \neq \tilde{r}_i \wedge b_i = 1$, i.e., unless WRONGRANDOM occurs.

Consider Step (vii). In the hybrid execution the honest machine M_P returns the triple $(\tilde{r}_i, s_i, b_i)$ where $(\tilde{r}_i, s_i)$ are computed according to the honest protocol and $b_i := \mathit{Verify}(P, n, s, \tilde{r}_i, q_1, \ldots, q_{t_2}, s_1, \ldots, s_i)$. In the ideal execution the honest machine $\tilde{M}_P$ returns the triple $(r_i, s_i, 1)$ (here r_i is the random value chosen by $\tilde{M}_P$ itself). Thus the triples returned in the hybrid and the ideal execution are equal unless $r_i \neq \tilde{r}_i \vee b_i \neq 1$. However, $r_i \neq \tilde{r}_i \vee b_i \neq 1$ implies WRONGPROOF $\vee$ WRONGRANDOM, so the triples returned in the hybrid and the ideal execution are equal unless WRONGPROOF or WRONGRANDOM occurs.

So together, we have that the view of the environment is identical in the hybrid and the ideal execution unless PREDICT, WRONGRANDOM, or WRONGPROOF occurs.

A.10 Putting the pieces together

We have seen so far that the real and the hybrid execution lead to the same outputs of H or $\tilde{H}$, respectively, unless BAD occurs. Thus in particular Z's output is the same unless BAD occurs. Furthermore, we have shown the same for the hybrid and the ideal execution. Therefore the output of Z is the same in the real and the ideal execution unless BAD occurs. Thus $|P_R - P_I| \leq \Pr_{\text{BAD}}$. Using the bound for $\Pr_{\text{BAD}}$ derived above, theorem 2 follows. $\square$