Limiting the impact of unreliable randomness in deployed security protocols

Liliya Akhmetzyanova*, Cas Cremers†, Luke Garratt‡, Stanislav Smyshlyaev§, and Nick Sullivan¶
*CryptoPro, Moscow, Russian Federation
Email: lah@cryptopro.ru
†CISPA Helmholtz Center for Information Security
Saarland Informatics Campus, Germany
Email: cremers@cispa.saarland
‡Cisco Meraki, San Francisco, United States
Email: lgarratt@cisco.com
§CryptoPro, Moscow, Russian Federation
Email: smyshsv@gmail.com
¶Cloudflare Inc., San Francisco, CA, USA
Email: nick@cloudflare.com

***Abstract*—Many cryptographic mechanisms depend upon the availability of securely generated random numbers. In practice, the sources of random numbers can be unreliable for many reasons, including bugs, compromise or subversion of standards. While there exist ways to significantly reduce the impact of unreliable randomness, these typically do not work well with practical constraints, such as long-term keys stored in hardware security modules. In practice, even modern protocols like TLS 1.3 lack such mechanisms and are therefore highly vulnerable to unreliable randomness.**

We propose a wrapper construction that reduces the impact of untrusted randomness, and which is is compatible with, and effective in, existing deployments of protocols such as TLS. We provide a security analysis of the construction and elaborate on design choices and practical interpretations. Our findings show that it is possible to effectively harden deployed protocols against unreliable randomness.

I. Introduction

Most key exchange protocols (e.g., TLS, SSH, and IKE) depend on the secure generation of random numbers. At the core of many of these protocols is a Diffie–Hellman key exchange of the following form: Alice generates a random number x, computes g^x and sends it to Bob. Bob similarly generates random number y, computes g^y and sends it to Alice. Of course, each particular protocol has essential additional details such as certificates, signatures, key derivation, comparison of transcripts and so on, but at the heart of all these protocols is a session key computed using the random numbers x and y and further material that is typically public. Therefore, it is essential that these random numbers are generated as securely as possible.

In many operating systems, raw entropy comes in the form of events such as mouse movements or keystrokes. As large amounts of raw entropy are difficult to accumulate, Alice and Bob do not generate truly random numbers x and y for their key exchanges. Instead, they use the primitive of a cryptographically secure pseudorandom number generator (CSPRNG), which takes as input a seed, continually harvests more raw entropy and produces arbitrarily many pseudorandom numbers as required. Since CSPRNGs are used to provide the essential secrets for the derived key, their security is critical. However, the current problem we face is that all CSPRNGs depend on raw entropy in some form or another. Therefore, if the underlying randomness is not good, the derived key is not secure.

It may be tempting to think that random number generation is well-understood by now. Unfortunately, this is not the case. There are many real-world examples of poor random number generation. We mention a few examples:

- The Debian OpenSSL random number generator vulnerability [2], [30], in which the range of the produced randomness was effectively much smaller than expected;
- Predictable random numbers in Android's Java OpenSSL [26] leading to theft of bitcoins;
- The Dual EC random number generator backdoor [6], by which observing produced random values could be used to infer the internal state of the generator for someone who knew the underlying relations, thereby making further values predictable. This caused several vulnerabilities including in TLS [8] and VPN routers [7];
- Random number generators on hardware that degrades over time;
- Servers with reliable long-term keys in HSMs that are deployed in settings where good randomness generation cannot be guaranteed, such as content delivery networks (CDNs) that deploy servers in remote locations to improve latency for local users;
- Internet of Things (IoT) devices that are deployed with potentially good (factory) keys, but have no access to good randomness generation mechanisms after deployment;
- Side channel attacks on random number generators to infer future states [9];
- Some AMD processors whose RDRAND stopped working

after suspend resume – reported as early as 2014 [25] but only addressed in 2019 [1], [24].

Given this situation, one would expect that modern security protocols provide some protection mechanisms against randomness that becomes unreliable over time. However, this is rarely the case: widely used protocols such as SSH [31], IKE [14], [16], and even the newest version of TLS (TLS 1.3 [29]) are highly vulnerable to unreliable randomness.

For example, the most common use case of TLS 1.3 [29] is based on a Diffie-Hellman exchange, in which the client and server each generate a fresh random value (e.g., x and y), and send each other the corresponding public key share (g^x, g^y) along with many other elements. The server signs its response, which includes its share g^y, using its long-term signing key; but both g^x and g^y are visible to anyone who can observe the network traffic, such as ISPs, backbone providers, rogue wifi access points, or routers. Then, a complex key derivation is used to compute the symmetric session keys that will be used to encrypt payload messages, but the only secret input to this key derivation is g^{xy} – all other inputs can be directly inferred from the network traffic. Hence, if an adversary knows or guesses either x (as generated by the client) or y (from the server), it can use the network traffic to compute g^{xy} and compute the session keys, and therefore to decrypt all subsequent TLS payloads or modify them.

The situation is similar for SSH [31], IKE [14], [16] (and therefore also IPSec [17]), and even for post-quantum proposals such as NewHope [3].

Related work

In the domain of key exchange protocols, at least since the MTI protocols [27] from 1986, there have been protocols that include both session-specific randomness and long-term keys into the computation of the session key. While this was not the objective of early protocols, an appropriate combination can strengthen the resilience of the resulting protocol against unreliable randomness. Such resilience has been considered, for example, in the extended analysis of the HMQV protocol [20]. The first time that unreliable randomness was explicitly considered in a Bellare-Rogaway-style security model for key exchange protocols, was in 2006 in [23]. In this work, the authors also proposed a variant of the Diffie-Hellman protocol called the NAXOS protocol, which is secure in their model. It relies on a construction sometimes referred to as the "NAXOS trick". In the standard Diffie-Hellman protocol, pseudorandom numbers x and y are computed from raw entropy and used as private exponents, and then publicly exchanging the messages g^x and g^y. Instead, the NAXOS protocol sends $g^{H(x,\mathsf{sk}_{\hat{A}})}$ and $g^{H(y,\mathsf{sk}_{\hat{B}})}$ where H is a hash function, and $\mathsf{sk}_{\hat{A}}$ and $\mathsf{sk}_{\hat{B}}$ are the long-term secret keys of Alice and Bob respectively. Intuitively we can see that security of the private exponent in this case should now depend on the pairs (x, $\mathsf{sk}_{\hat{A}}$) and (y, $\mathsf{sk}_{\hat{B}}$) being unknown to the adversary, as opposed to just x and y. Furthermore, a protocol that uses $g^{H(x,\mathsf{sk}_{\hat{A}})}$ can behave in exactly the same way as one that uses g^x. The main difference is that $H(x, \mathsf{sk}_{\hat{A}})$ is harder to learn for the adversary than just x.

However, in the case of implementing cryptographic protocols in a hostile environment, often the long-term term sk is stored in trusted hardware that does not provide direct access to compute $H(x, \mathsf{sk})$. For many deployments that use hardware security modules (HSMs), this prevents the application of the NAXOS trick. Furthermore, it is generally undesirable to re-use signing keys as input to a hash function from a cryptographic design perspective.[1]

Besides the weaknesses on random number generators mentioned earlier, there is a rich literature on exploiting various subtleties of random number generators and their implementation details. There are many examples of PRGs that were not properly seeded, including SecureRandom [18] and other Java PRGs [28], leading to situations such as widespread weak long-term TLS keys [15]. Along the same lines, hard-coded seed keys for the ANSI X9.31 RNG lead to state recovery attacks [10] enabling the decryption of, e.g., FortiOS VPN traffic.

Contribution

Inspired by the NAXOS trick, we propose a "wrapper" function around existing pseudorandom number generators, in contexts that have access to a signing algorithm for the long-term key of a party. Our construction uses a signature $\mathsf{Sig}(\mathsf{sk}, \mathtt{tag}_1)$ for some fixed string $\mathtt{tag}_1$ instead of using a secret key sk.

In contrast to the NAXOS trick, our design is more generic and applies outside of the Authenticated Key Exchange (AKE) domain, and we follow a much more conservative approach that enables the re-use of existing infrastructure. Our construction furthermore offers improved graceful degradation if it turns out that the output of the hash function reveals partial information about the input. The proposed construction is currently considered in IETF (CFRG), see [11] for details.

The construction aims to provide the following informal security properties:

1) If the CSPRNG works as expected, that is, in a certain security model the CSPRNG output is indistinguishable from a truly random sequence, then the output of the proposed construction is also indistinguishable from a truly random sequence in that security model.
2) If the CSPRNG is broken or controlled by the adversary, then the output of the proposed construction remains indistinguishable from random, provided that the private key remains unknown to the adversary.
3) An adversary with full control of a (potentially broken) CSPRNG that is also able to observe all outputs of the proposed construction, has no non-negligible advantage in leaking the secret key, modulo side channel attacks.

[1]One could imagine deploying a secondary secret, but this is even less compatible with existing deployments, as it would require globally distributing a secondary secret to existing devices.

We define the formal security model regarding these three properties in terms of a game between a challenger and an adversary. We provide proofs that the proposed wrapper construction meets the precisely defined security guarantees under standard cryptographic assumptions.

Our construction can be transparently used with existing security protocols such as TLS, IKE, and SSH. Because it only requires local code changes at the interface between protocol and randomness source, it does not lead to externally visible changes. Our construction is therefore backwards compatible by design, and can be rolled out incrementally. Concretely, this means that it is possible to update for example some TLS 1.3 servers with our construction and continue operation, without needing to modify the clients. This would already increase security guarantees for those servers as well as the unmodified clients communicating with them.

As we will show later, our construction requires the generation of one signature at startup time, and negligible cost during normal protocol operations.

Organization

In Section II we introduce our construction for randomness improvement. In Section III we define the security properties we will need for our primitives in our wrapper construction. In Section IV we define the security model for our wrapper construction and additionally provide a practical interpretation. In Section V we give a security theorem and formal game hopping security proof that our construction fulfils the claimed security properties. In Section VI we provide experimental results to assess the efficiency of our construction.

II. The wrapper construction

Our objective is to reduce the impact of bad randomness on security protocols. We assume that we are given a function G, which is intended to be a CSPRNG, but for which we cannot be sure if it always produces good randomness, or if it was subverted. One way of reducing the impact, as used by the NAXOS protocol, is to ensure that (i) a party's long-term private key sk must be known in order to produce the final random values, and (ii) observing a random value cannot be used to predict the next value, even if the underlying CSPRNG might be vulnerable to state recovery.

We first present our construction, and then discuss the design constraints. Let $\mathsf{Sig}(\mathsf{sk}, m)$ be a deterministic function that computes a signature of message m given private key sk. Let H be a cryptographic hash function that produces output of length M. Let $\mathsf{KDF}(x, y)$ be a key derivation function, which accepts a random key x and a salt y and produces a pseudorandom key of length L. Let $\mathsf{PRF}(k, c, n)$ be a variable-length output pseudorandom function, that takes as input a pseudorandom key k of length L, info string c, and output length n, and produces output of length n.

Let $G(n)$ be an algorithm that generates n random bits, i.e., the output of a CSPRNG. Instead of using $G(n)$ when randomness is needed, use an augmented CSPRNG $G'(n)$, where

$$G'(n) = \mathsf{PRF}(\mathsf{KDF}(\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \mathtt{tag}_1)), y), \mathtt{tag}_2, n),$$

and where $y = G(L)$, $\mathtt{tag}_1$ is a fixed, context-dependent string, and $\mathtt{tag}_2$ is a dynamically changing string (e.g., a counter increased on each invocation) of M bits. We require that $L \geq n - M$.

The wrapper construction is depicted graphically in Figure 1. In the figure, "*next*" represents the wrapper instance invocation, which increases the value of the counter used for $\mathtt{tag}_2$.

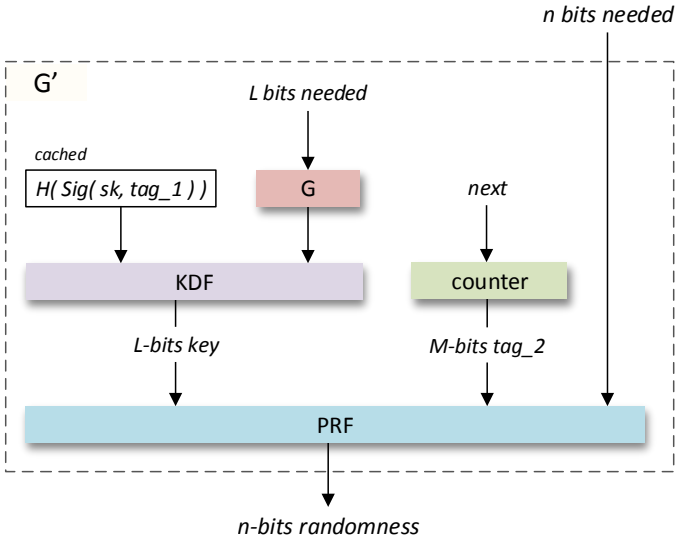

Fig. 1: The wrapper construction.

Design choices

We give a brief overview of the reasons that lead to the specific design of the construction.

First, our goal is to provide security guarantees for vulnerable environments that are compatible with existing deployments. This leads to two requirements: (i) for compatibility and practical deployment, we would like to re-use existing long-term keys, and (ii) in vulnerable environments, long-term keys are likely to reside in trusted hardware. These requirements are typically incompatible with the NAXOS design, both for cryptographic separation as well as the limitations of trusted hardware APIs. We therefore need to be able to use sk in some trusted hardware, which implies that we need to choose an operation and some data for this operation. In the vast majority of real-world systems, private keys are allowed to be used for digital signatures by the APIs of trusted hardware. Our wrapper therefore assumes from the context that there exists a signature key sk that can be used by the wrapper for signing.

Since signature operation is computationally expensive, we design our wrapper to avoid producing a signature per randomness request. To prevent collisions of wrapper outputs it is highly desirable to bind all information connected to the instance of the CSPRNG together as early as possible, i.e. right

when using the key. For this reason, we sign a specific tag $\texttt{tag}_1$, which is a constant string bound to a specific device and protocol, incorporating all available information specific to the process and having the format that is not supported (or explicitly forbidden) by other applications using sk. Therefore, we have a signature over a unique-per-instance data, that can be used multiple times for all requests to the instance of the wrapper.

We require Sig to be a deterministic signature function or to use an independent entropy source, since broken randomness used in a randomized signature scheme can lead to the total compromise of the private key. This is explicitly something we need to avoid (property three). We also hash the signature output to have additional protection for the signature itself, and thus the private key sk. Another advantage of hashing is that there is no guarantee that the signature is uniformly distributed across the range of Sig. Moreover, after hashing we can obtain a shorter intermediate value, which is good for overall performance. Crucially, this value can be cached, since it remains the same for all requests to one wrapper instance. This is an important design choice that ensures our construction is sufficiently efficient.

Now we need to mix the obtained value of $\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1))$ together with the initial output of the CSPRNG and provide the first and the second declared security properties: if either CSPRNG works fine or sk remains secret, then the output of the wrapper must be good. To achieve this, we first "extract" an intermediate pseudorandom key (PRK) from the values of $\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1))$ and y – the two values that are normally assumed to be secret – using a KDF function. The output is then provided as the input to a PRF function, together with the original (potentially unreliable) randomness and a unique (non-secret) value $\texttt{tag}_2$, which changes upon each invocation of the wrapper.

There are many choices for instantiating the KDF and PRF functions in our construction. One reasonable choice is to rely on a well-studied construction such as HKDF [19], [21], by using HKDF-Extract as the KDF function and HKDF-Expand as the PRF function. HKDF is standardised [19], well studied, and likely to already be implemented in many of the potential deployment environments, thereby reducing the amount of additional code needed.

Implementing $\texttt{tag}_2$ as a counter in the construction has the additional benefit of producing unique pseudorandom numbers, even if the source randomness becomes constant (e.g., always zero). It additionally yields a larger output range even if the range of the original randomness is too small, e.g., as happened in the Debian RNG case [30], [2]. This design aspect is inspired by the observations and constructions from [13], and in particular by their CNX protocol, which introduced a counter for the same purpose, but formulated and analysed in the context of key exchange protocols.

III. Base security definitions

We define the security properties we need of our primitives for our construction to be proven secure in our security model. We address the key derivation, the pseudorandom function, and the signature scheme in turn below.

A. Key derivation function

A key derivation function is an algorithm KDF that implements a deterministic function $k = \mathsf{KDF}(x, y)$, taking as input some bit strings x and y, and returning a key $k \in \{0,1\}^L$. We require that KDF fulfills the following two security properties.

1) KDF security. Intuitively, this property means computational indistinguishability of the $\mathsf{KDF}(x, \cdot)$ function for x chosen uniformly at random from an ideal random function $\rho(\cdot)$. Namely, it is the standard PRF security property. Let ϵ_{KDF} denote the probability that any probabilistic polynomial time adversary is able to distinguish between these distributions.

2) Uniformness preserving. We require the KDF algorithm to fulfill the uniformness preserving of the output for $y \leftarrow_{\$} \{0,1\}^L$ chosen uniformly at random and x chosen arbitrarily. We define a security game between an adversary A and a challenger as follows.

1) The adversary makes a query x. The challenger uniformly randomly samples a secret value $y \leftarrow_{\$} \{0,1\}^L$. The challenger computes $k_0 := \mathsf{KDF}(x, y)$ and samples $k_1 \leftarrow_{\$} \{0,1\}^L$ uniformly randomly. The challenger then flips a coin $b \leftarrow_{\$} \{0,1\}$ and responds with k_b.
2) The adversary then outputs a guess b' for b and wins the game if $b = b'$ and loses otherwise.

Let $\epsilon_{\texttt{UP}}$ denote the probability that any probabilistic polynomial time adversary is able to win the considered above game. In other words, for any probabilistic polynomial time algorithm A,

$$\left| \Pr(b = b') - \frac{1}{2} \right| \leq \epsilon_{\texttt{UP}}.$$

The KDF algorithm is said to be $\texttt{UP}$-secure if $\epsilon_{\texttt{UP}}$ is negligible in the security parameter.

HKDF-extract. In our concrete construction, we instantiate KDF with HKDF-extract [22], which is literally the HMAC function.

The KDF property and the uniformness preserving property follow directly if the KDF function is a PRF is a so-called *dual-PRF*: i.e., it is a PRF even when swapping the inputs x and y). This is a standard assumption for HMAC and HKDF that was introduced by Bellare [4], in which the security of HMAC is proven by assuming the dual PRF property of the compression function. The PRF property of HMAC when keyed by the first input follows from to the security analysis (see, e.g. [4], [5]). Moreover, for the wrapper, the length of the HMAC key $x = \mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1))$ is equal to the hash output length, thereby satisfying the recommendations from these analyses. The dual-PRF assumption then ensures that the PRF property of HMAC also holds when keyed by the second input and commonly used as a standard assumption (See, e.g., in [12]).

B. Variable-length output pseudorandom function

A variable-length output pseudorandom function is an algorithm PRF that implements a deterministic function $z =$

$\mathsf{PRF}(k, c, n)$, taking as input a key $k \in \{0,1\}^L$, some bit string c, an integer n, and returning a string $z \in \{0,1\}^n$. We assume that the maximum permitted size n is polynomial.

We define a security game (for multi-user setting) between an adversary A and a challenger as follows.

1) The challenger uniformly randomly samples q (polynomial) secret keys $k_1, \ldots, k_q \in \{0,1\}^L$ independently from each other.
2) The adversary is allowed to query the challenger with adaptively chosen values (i, c, n), $i \in [1, \ldots, q]$. The challenger replies with $\mathsf{PRF}(k_i, c, n)$.
3) Eventually the adversary queries a special symbol T to indicate the so-called test query with value (i, c, n) where $(i, c, \cdot)$ was not queried before. At this point, the challenger computes $z_0 \coloneqq \mathsf{PRF}(k_i, c, n)$ and samples z_1 uniformly randomly from $\{0,1\}^n$. The challenger then flips a coin $b \leftarrow_\$ \{0,1\}$ and responds with z_b.
4) The adversary is allowed to keep making queries. Note that after the test query the adversary is not allowed to query $(i, c, \cdot)$.
5) The adversary then outputs a guess b' for b and wins the game if $b = b'$ and loses otherwise.

Let $\epsilon_{\mathtt{mu-PRF}}$ denote the probability that any probabilistic polynomial time adversary is able to win the considered above game. In other words, for any probabilistic polynomial time algorithm A,

$$\left| \Pr(b = b') - \frac{1}{2} \right| \leq \epsilon_{\mathtt{mu-PRF}}.$$

The PRF algorithm is said to be $\mathtt{mu}$–PRF-secure if $\epsilon_{\mathtt{mu-PRF}}$ is negligible in security parameter.

C. Hash function and signature scheme

A signature scheme is a triple $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$. KGen is a probabilistic algorithm which takes as input the security parameter 1^α and outputs a public signature verification key pk and secret signing key sk. Sig is a signing algorithm which generates a signature σ for message m using secret key sk. In this work we consider only deterministic Sig algorithms. Vf is a deterministic signature verification algorithm which, given input (pk, σ, m), outputs 1 if σ is a valid signature of m under key pk, and 0 otherwise. It is required that for every k, every $(\mathsf{sk}, \mathsf{pk})$ output by $\mathsf{KGen}(1^\alpha)$, and every message m, it holds that

$$\mathsf{Vf}(m, \mathsf{Sig}(\mathsf{sk}, m)) = 1.$$

In our construction, we will not want to use the long-term key sk directly. We instead use the hash of a signature of $\mathtt{tag}_1$, signed with sk. For our security proof, we require that the combined hash function and signature scheme fulfill the following security property.

Definition 3.1 (Sig, H*-secure):* Consider the following game between a challenger and a polynomial time adversary A:

1) The challenger generates a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ using KGen and gives the adversary the public key pk.
2) The adversary is allowed to adaptively query chosen messages m to the challenger. The challenger responds to each query with $\sigma = \mathsf{Sig}(\mathsf{sk}, m)$.
3) When the adversary decides to, it outputs a so-called test query m^* that was not queried before. At this point, the challenger flips an unbiased coin $b \leftarrow_\$ \{0,1\}$. If heads, it returns with $\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, m^*))$. If tails, it responds with a uniformly random string of the same length.
4) The adversary is allowed to keep asking for signatures with messages $m \neq m^*$, but eventually it must output a guess b' for the coin flip b, at which point the game ends. The adversary wins if it guesses the coin flip correctly and loses otherwise.

Let $\epsilon_{\mathsf{Sig,H}}$ denote the probability that any probabilistic polynomial time adversary is able to win the considered above game. In other words, for any probabilistic polynomial time algorithm A,

$$\left| \Pr(b = b') - \frac{1}{2} \right| \leq \epsilon_{\mathsf{Sig,H}}.$$

The $(\mathsf{Sig}, \mathsf{H})$ composition is said to be Sig, H-secure if $\epsilon_{\mathsf{Sig,H}}$ is negligible in the security parameter.

This security property can be instantiated with routine cryptographic assumptions such as a hash function (e.g. BLAKE or SHA-3) modelled as a random oracle and an existentially unforgeable signature scheme (e.g. deterministic ECDSA).

Proof sketch: the proof uses the idea that (due to the random oracle properties) the polynomial time algorithm can only win the hash-signature game with non-negligible probability if the algorithm makes the query $\mathsf{Sig}(\mathsf{sk}, m^*)$ to the random oracle. Based on this idea, we can construct an adversary in the standard euf-cma game that simulates the hash-signature game and makes a forgery as follows.

All routine signing queries are transmitted by the adversary to its challenger. The random oracle is simulated by sampling a uniformly random string of the appropriate length for every new query, and by saving these input/output values locally. Without loss of generality, we assume that all queries are pairwise different.

The test query with a message m^* and the random oracle queries are processed as follows. For the test query the adversary checks if there is a valid signature for m^* among the queries already made to the random oracle. If there is, the adversary made a forgery and won its game. Otherwise, the adversary responds with a uniformly random string of the appropriate length and save it. Further queries to the random oracle are also checked for validity for m^*. If one of them is successfully tested, then the adversary made a forgery and won the game.

Minor discussion. Using a random oracle is a common way to ease the cryptographic analysis by making it modular. However, one should always keep in mind that a random oracle cannot be instantiated by any real hash function and, therefore, one should use it very carefully, trying to interpret the obtained security results. In our case, if the (Sig, H) composition turn

out to be insecure, then, due to the proof sketch, the used signature scheme is broken or the used hash function does not sufficiently disrupt the link between the domain and the range.

The crucial point of using a random oracle is probabilistic independence of its output values from any other values. Therefore, one should not to use the same hash function as used, e.g., inside the signature scheme. The common way to deal with it using one hash function it is to add some unique constant prefix before hashing.

IV. Security model

In this section we define the security property that we expect from our wrapper in terms of a game between a challenger and an adversary. Intuitively, we aim to prove that the wrapper does not degrade security and offers additional security in cases where the current randomness is unreliable.

Let $\mathcal{T}_1$ denote the set of possible values for $\mathtt{tag}_1$, and $\mathcal{T}_2$ the set of possible values for the dynamic tag $\mathtt{tag}_2$.

We run a `Wrapper` game between the challenger and the adversary as follows. At the beginning of the game, the challenger uniformly randomly chooses a secret key sk and returns a public key pk to the adversary. After receiving the public key, the adversary chooses a set $T \subseteq \mathcal{T}_1$ consisting of l pairwise different values $t_1, \ldots, t_l$ and sends it to the challenger. We assume that the size l of the set T is polynomial. The adversary is allowed to make the following queries to the challenger.

- The adversary can make `output` queries of two types:
  - $\mathtt{tag}_1, \mathtt{tag}_2, n$ (in this case the challenger chooses the $y \leftarrow\!\!\$\, \{0,1\}^L$ value uniformly randomly by itself);
  - $\mathtt{tag}_1, \mathtt{tag}_2, y, n$ (in this case the $y \in \{0,1\}^L$ value is chosen by the adversary).

  Note that the adversary is allowed to make queries where $\mathtt{tag}_1 \in T$. We also assume that the adversary does not make trivial `output` queries containing the same tuple $(\mathtt{tag}_1, \mathtt{tag}_2, y)$.
  The challenger produces

  $$\mathsf{PRF}(\mathsf{KDF}(\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \mathtt{tag}_1)), y), \mathtt{tag}_2, n)$$

  and returns this value as a response to the corresponding `output` query.
  The challenger indexes queries and locally saves the corresponding inputs to the wrapper for each query, i.e. the challenger saves records of the form $(i, \mathtt{tag}_1^i, \mathtt{tag}_2^i, y_i)$. We assume that $i \leq q$ for some q that is also polynomial.
- The adversary can make `sign` queries for signatures with sk of messages m. The challenger produces and returns the value $\mathsf{Sig}(\mathsf{sk}, m)$.
- The adversary can make a `corrupt` query for sk at any time. The challenger responds to this query with sk.
- The adversary is also allowed to make `reveal` queries for the y_i value used in the ith `output` query at any time (the adversary makes the query with the index of the target `output` query). The challenger must respond with the y_i value used in generating the response to the ith `output` query.
- At some point in time, the adversary must make a so-called test `output` query. This is the same as a normal non-trivial `output` query except the challenger flips an unbiased coin and either responds with the genuine output using the wrapper, or a uniformly randomly chosen string of the same length.
- The adversary is allowed to query for the y used in the test if it wants to, as well as to continue making other queries. The adversary may also query for the sk value or for the signature of $\mathtt{tag}_1$ if it has not already done so. We impose a restriction on the adversary, which we encode in a so-called *freshness* condition specified below. During the game, we require that the test output query must remain "fresh" at all times.
  Freshness: We say that the ith `output` query is "fresh" if one of the values y_i or $\mathsf{Sig}(\mathsf{sk}, \mathtt{tag}_1^i)$ remains secret. That is, if one of the following conditions is satisfied:
  - The adversary has not made the `corrupt` query or the `sign` query for $m = \mathtt{tag}_1^i$.
  - The adversary has not made the `reveal` query for y_i.
- At some point in time after the test query, the adversary must output a single bit as a guess. The adversary wins the game if it is able to guess the coin flip with non-negligible advantage over $\frac{1}{2}$.

The above security model is also presented in pseudocode in Figure 2.

Motivation for the security model

In this section we provide the arguments why the proposed security model correctly captures our desired properties, as stated informally in Section I on page 2.

Consider the second informal security property as stated in the contributions, considering the adversaries which are able to break or even to control the CSPRNG output. The proposed model captures these capabilities by allowing the adversary to make the second type `output` queries with the CSPRNG output chosen by itself and to `reveal` the CSPRNG outputs chosen by the challenger uniformly at random.

Note that the model also considers the case when the CSPRNG works as expected but the private signing key or signature is compromised (see the first informal security property). This case is captured by allowing the adversary to make the first type `output` queries with strong initial randomness chosen by the challenger and to `corrupt` a private key. The first type `output` queries implicitly assume the CSPRNG to be secure and securely used during the queries processing.

The security model and the security proof depend on the concept of "freshness", which intuitively encodes the adversary not winning the game trivially by making the obvious `reveal` or `sign`/`corrupt` queries (see definition of *fresh* queries). This freshness restriction only prevents the adversary from knowing *both* the signature and the output of G at the same time, in which case it can compute the output of the wrapper by guessing $\mathtt{tag}_2$; all other cases, e.g., where one of these values

$\mathsf{Exp}^{\mathsf{Wrapper}}_{G'}(\mathcal{A})$:

$b \leftarrow_{\$} \{0,1\}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^{\alpha})$
$(T, state) \leftarrow_{\$} \mathcal{A}(\mathsf{pk}, \mathtt{init})$
$seq \leftarrow 0,\ tested \leftarrow 0$
$Chosen \leftarrow \emptyset,\ Signed \leftarrow \emptyset$
$Y \leftarrow [\,]$
$corrupted \leftarrow \mathsf{false}$
$revealed \leftarrow \mathsf{false}$
$fresh \leftarrow \mathsf{true}$

.....Setup completed.....

/ Running the adversary $\mathcal{A}$
$b' \leftarrow_{\$} \mathcal{A}^{\mathsf{Output},\mathsf{Reveal},\mathsf{Sign},\mathsf{Corrupt}}(\mathsf{pk}, state)$
if $\big((tested \in Chosen) \vee revealed = \mathsf{true}\big) \wedge$
$\big((\mathtt{tag}_1 \in Signed) \vee corrupted = \mathsf{true}\big)$
then
$fresh \leftarrow \mathsf{false}$

.....Finalization completed.....

return $b = b' \wedge fresh = \mathsf{true}$

$\mathsf{Output}(\mathtt{tag}_1, \mathtt{tag}_2, y = \varepsilon, n, test),\ \mathtt{tag}_1 \in T$:

$seq \leftarrow seq + 1$
if $y = \varepsilon$ **then**
$y \leftarrow_{\$} \{0,1\}^L$
else
$Chosen \leftarrow Chosen \cup \{seq\}$
$k \leftarrow \mathsf{KDF}(\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \mathtt{tag}_1)), y)$
$out \leftarrow \mathsf{PRF}(k, \mathtt{tag}_2, n)$
$Y[seq] \leftarrow y$
/ One time test query
if $test = \mathsf{true}$ **then**
$tested \leftarrow seq$
if $b = 0$ **then**
$out \leftarrow_{\$} \{0,1\}^n$
return out

$\mathsf{Reveal}(i), 0 < i \leqslant seq$:

if $i = tested$ **then**
$revealed \leftarrow \mathsf{true}$
return $Y[i]$

$\mathsf{Corrupt}()$:

$corrupted \leftarrow \mathsf{true}$
return sk

$\mathsf{Sign}(m)$:

$Signed \leftarrow Signed \cup \{m\}$
return $\mathsf{Sig}(\mathsf{sk}, m)$

Fig. 2: Security experiment for adversary $\mathcal{A}$ against security of wrapper G'.

is known to the adversary, are considered by the analysis. Thus, the security proof obtained in this model can be interpreted as follows. If the adversary learns only one of the signature or the usual randomness generated on one particular instance, then under the security assumptions on our primitives, the wrapper construction should output randomness that is indistinguishable from a random string. Together, these two cases cover the first and second informal properties. The third property, regarding the confidentiality of the private key, follows directly from the guarantees of the signature scheme.

Our security model also explicitly assumes that $\mathsf{Sig}(\mathsf{sk}, \mathtt{tag}_1)$ never appears externally elsewhere in the protocol so that the adversary has no hope of seeing it and using it. If this dedicated signature becomes public, security degrades to the normal case of generating pseudorandom numbers. Of course, in practice actually forcing $\mathsf{Sig}(\mathsf{sk}, \mathtt{tag}_1)$ to appear may still be difficult or impossible for an adversary. Note also that these signatures are agent-specific, which improves containment. Thus, leaking $\mathsf{Sig}(\mathsf{sk}, \mathtt{tag}_1)$ only for a specific agent, using a unique fixed string $\mathtt{tag}_1$, only affects this agent, and has no consequences for other agents using the same key sk. The fact that in reality $\mathtt{tag}_2$ changes also helps to prevent the use of repeated random numbers when there is a poor entropy source.

The tags are assumed to be known and even to be chosen by the adversary in our security model. In practice they may not be, which only adds an extra layer of protection in practice. The only restrictions on the choice is that the tags values along with the CSPRNG output should be unique in total for the test query.

Note that the model also restricts the size of the set T containing the possible values of $\mathtt{tag}_1$ for the security proof. In practice this restriction means that the number of agents using the same secret key sk must be negligible in the security parameter, and that the set of $\mathtt{tag}_1$, intended to be used during the long-term key life, should be fixed at the system initialization stage. Obviously, both can be easily achieved in reality.

V. Proof of Security

Theorem 5.1: Let PRF be a Variable-length output pseudo-random function, and let KDF be a key derivation function that is KDF secure and uniformness preserving. Let Sig be a deterministic signature and H a hash function that together are Sig, H-secure. Then, any probabilistic polynomial time adversary has only negligible advantage in winning the security game described in Section IV.

Proof 5.2: We prove the theorem via a game hopping proof: Starting from the base challenge game, we apply

small variations that yield negligible advantage (based on our primitive security assumptions) and ultimately bound to a probability negligible in the security parameter.

Let Game 0 denote the original security game as defined in our security model definition. Let S_j denote the event of the adversary winning Game j. Our goal in this proof is to bound $\Pr(S_0)$ to show that it is only at most negligibly above $\frac{1}{2}$. Although the security argument is very intuitive ("the adversary must surely need both y and sk to guess the secret") we will formally prove it using a game hopping proof.

At some point in time, the adversary must issue a $\texttt{test}$ query. Let $\texttt{tag}_1^i, \texttt{tag}_2^i, y_i$ denote the values used by the challenger for this query. We prove this theorem in a case partition on whether the adversary has revealed $\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1^i)$ or y_i (if the adversary has queried for neither, then it is clearly in an even worse position to win the game).
We now proceed with our case partition.

Case 1: The adversary has revealed y_i or has chosen y_i by itself.: In this case, the adversary is not allowed to query for sk or for the signature of $\texttt{tag}_1^i$, otherwise the test output query would not be fresh.

Game 1. Let Game 1 be identical to Game 0 except the challenger guesses in advance an integer $j \in [1, \ldots, l]$ and aborts (and the adversary loses) unless $\texttt{tag}_1^i = t_j$. This is a large failure event game hop and it is easy to see that

$$\Pr(S_0) \leq l \Pr(S_1).$$

Game 2. Define Game 2 to be identical to Game 1 except $\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1^i))$ is replaced with a value x_i sampled uniformly at random for each $\texttt{output}$ query with $\texttt{tag}_1 = \texttt{tag}_1^i$. Here we claim that

$$\Pr(S_1) \leq \Pr(S_2) + \epsilon_{\mathsf{H},\mathsf{Sig}}.$$

In particular, no probabilistic polynomial time distinguisher algorithm can distinguish between Game 1 and Game 2, since this would imply a way to beat the hash-signature game with better than $\epsilon_{\mathsf{H},\mathsf{Sig}}$ advantage. Precisely, an adversary in the hash-signature game could beat it with better than $\epsilon_{\mathsf{H},\mathsf{Sig}}$ advantage as follows.

It acts as a challenger in the hybrid game and inserts the value of pk from the hash-signature game. As an adversary in the hash-signature game, it inserts the known $\texttt{tag}_1$ value from the previous game and asks a test query straight away, receiving either $\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1^i))$ or a uniformly randomly chosen value x_i as per the rules of the hash-signature game. It inserts this value as the value for $\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1^i))$ in the hybrid game. Based on Game 1, the adversary knows where to make this swap in the game. The $\texttt{output}$ queries with $\texttt{tag}_1 \neq t_j$ are processed by the adversary using queries for signatures to its challenger. The adversary can now simulate $\texttt{output}$ and test queries as normal using this value. The adversary does not have to worry about simulating a $\texttt{corrupt}$ query because we are in case 1. Finally, it can simulate signature queries by merely forwarding them along to the hash-signature game. This perfectly simulates the hybrid game: it is identical to Game 1 if the test returns $\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1^i))$, and it is identical to Game 2 if it returns a uniformly randomly chosen value x_i. The adversary follows the coin flip choice in the simulated hybrid game as its guess for the hash-signature game. By assumption of the hardness of the hash-signature game, the adversary can only win with at most advantage $\epsilon_{\mathsf{H},\mathsf{Sig}}$. Therefore, the difference in advantages of adversaries across Game 1 and Game 2 can also only be separated by at most $\epsilon_{\mathsf{H},\mathsf{Sig}}$. This completes the proof of the above claim.

Game 3. Define Game 3 to be identical to Game 2 except $\mathsf{KDF}(x_i, \cdot)$ is replaced with an ideal random function $\rho(\cdot)$ for $\texttt{output}$ query with $\texttt{tag}_1 = t_j$. We claim that

$$\Pr(S_2) \leq \Pr(S_3) + \epsilon_{\mathsf{KDF}}.$$

Here we present the simulation argument for the KDF game. Because of Game 1, the adversary knows where to make this swap in the game precisely for $\texttt{output}$ queries with $\texttt{tag}_1 = t_j$. The KDF adversary makes queries y to compute $\mathsf{KDF}(x_i, y)$ for required y for the key x_i chosen by the challenger uniformly at random. If the adversary has made the $(\texttt{tag}_1, \texttt{tag}_2, n)$ query (without y), then the KDF adversary chooses y value uniformly randomly by itself and locally saves it. Note that it can answer the reveal query in this case. All other queries are simulated in the normal way. This perfectly simulates the hybrid game: it is literally Game 2 if the KDF challenger provides outputs of the KDF function, and it is literally Game 3 if the challenger provides outputs of the ideal random function ρ. The adversary follows the coin flip choice in the simulated hybrid game as its guess for the KDF game. By assumption of the hardness of the KDF game, all adversaries can only win with at most advantage ϵ_{KDF}. Therefore, the difference in advantages of adversaries across Game 2 and Game 3 can also only be separated by at most ϵ_{KDF}. Thus the claim above is proven.

Game 4. Define Game 4 to be identical to Game 3 except the response to the test $\texttt{output}$ query is replaced with a value chosen uniformly randomly.

Here we claim that

$$\Pr(S_3) \leq \Pr(S_4) + \epsilon_{\texttt{mu-}\mathsf{PRF}}.$$

Here we present the simulation argument for the multi-user PRF game with at most q keys where q is the number of $\texttt{output}$ queries. Because of Game 1, the PRF adversary knows what $\texttt{output}$ queries should be processed with the usage of its challenger. The $\texttt{output}$ queries with $\texttt{tag}_1 = t_j$ and different y are processed with the different keys by asking the mu-PRF challenger with the suitable indexes in queries. Since the test query should be non-trivial, y_i or $\texttt{tag}_2^i$ should be new (we neglect the probability that for the $\texttt{output}$ query of the type $(\texttt{tag}_1, \texttt{tag}_2, n)$ the y_i value chosen by the challenger itself collides with the previous values). Therefore, the mu-PRF adversary can use its test query to swap the $\mathsf{PRF}(\rho(y_i), \texttt{tag}_2^i, n)$ with the value obtained as a response on the test query in the mu-PRF game. All other queries are simulated in the normal way. This perfectly simulates the hybrid game. Therefore, the difference in advantages of adversaries

across Game 2 and Game 3 can also only be separated by at most $\epsilon_{\texttt{PRF}}$. Thus the claim above is proven.

It is clearly impossible for the adversary to have any advantage in guessing the secret bit in Game 4 (in either case, the wrapper generates a uniformly randomly chosen string). As such, $\Pr(S_4) = \frac{1}{2}$. Thus in this case $\Pr(S_0) \leq \frac{1}{2} + l(\epsilon_{(\mathsf{H},\mathsf{Sig})} + \epsilon_{\mathsf{KDF}} + \epsilon_{\texttt{mu-PRF}})$ which is negligibly close to $\frac{1}{2}$.

Case 2: The adversary has revealed sk *or* $\mathsf{Sig}(\mathsf{sk}, tag_1^i)$*.:* In this case, the adversary is not allowed to reveal y_i or to choose y_i by itself, otherwise the test output query would not be fresh. Thus, $y_i \in \{0,1\}^L$ is chosen uniformly randomly by the challenger and is used only for the test `output` query.

Game 1. This game is identical to Game 0 except $\mathsf{KDF}(\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1^i)), y_i)$ for test `output` query is replaced with a value k sampled uniformly at random.

We claim that

$$\Pr(S_0) \leq \Pr(S_1) + \epsilon_{\texttt{UP}}.$$

In the simulation the UP adversary for KDF chooses private key sk by itself and simulates all queries except for the test `output` query in the normal way. To simulate the test `output` query $(\texttt{tag}_1^i, \texttt{tag}_2^i, n)$ the UP adversary makes a query with $\mathsf{H}(\mathsf{Sig}(\mathsf{sk}, \texttt{tag}_1^i))$ to its UP challenger. The challenger chooses $y_i \leftarrow\!\!\$\, \{0,1\}^L$ and returns a value k_b. Then the UP adversary computes and returns the value $\mathsf{PRF}(k_b, \texttt{tag}_2^i, n)$. This perfectly simulates the hybrid game. Therefore, the difference in advantages of adversaries across Game 0 and Game 1 can also only be separated by at most $\epsilon_{\texttt{UP}}$. Thus the claim above is proven.

Game 2. Define Game 2 to be identical to Game 1 except the response to the test `output` query is replaced with a value chosen uniformly randomly.

Here we claim that

$$\Pr(S_1) \leq \Pr(S_2) + \epsilon_{\texttt{mu-PRF}}.$$

The simulation is the same as for Game 4 from the other case but here is enough to make one test query for one key in the multi-user PRF game since the key k is used once for the test `output` query only.

It is clearly impossible for the adversary to have any advantage in guessing the secret bit in Game 2 (in either case, the wrapper generates a uniformly randomly chosen string). As such, $\Pr(S_2) = \frac{1}{2}$. Thus in this case $\Pr(S_0) \leq \frac{1}{2} + \epsilon_{\texttt{UP}} + \epsilon_{\texttt{mu-PRF}}$ which is negligibly close to $\frac{1}{2}$.

VI. Performance

To evaluate the performance of our wrapper construction, we implemented the randomness wrapper in C using the BoringSSL library. We implemented the Extract and Expand routines using HKDF-Extract and HKDF-Expand, as described in [19]. In particular, we used SHA-256 as the hash algorithm.

The computational cost of our construction comes in two parts: the first is the one-off cost of computing the hashed signature, which is the same for all invocations, and therefore can be cached. This amounts roughly to producing one signature at initialisation/startup time. For any concrete implementation this is negligible compared to other initialisation/startup operations.

The second part is the running cost, i.e., the overhead during subsequent generation of random numbers. Our experiments show that while the additional cost is significant compared to plain random number generation, it is ultimately dwarfed by the computational cost of typical protocols that would employ our construction, ultimately generating negligible overhead. We describe our experiments to support these conclusions in turn.

A. Isolated wrapper overhead

With our implementation, we then ran the following baseline and wrapper experimental algorithms detailed in Algorithm 1 and 2, respectively, with configuration inputs sk, $\texttt{tag}_1$, and $\texttt{tag}_2$ of 64 bits, and requested output size of $n = 256$ bits corresponding to 32 bytes, loop timer $T = 1\text{E} + 09$ (in nanoseconds, corresponding to 1.0s), and KDF output size of $L = 256$ bits. The wrapper experimental algorithm computes the signature *once* and reuses it for all subsequent extractions. Now is a function that returns the current time in nanoseconds.

Algorithm 1 Baseline Experimental Algorithm

```
C ← 0
t_start ← Now()
t_end ← t_start + T
while Now() < t_end do
    r ← G(n)
    C ← C + 1
end while
return C
```

Algorithm 2 Wrapper Experimental Algorithm

```
s ← cached value for Hash(Sign(sk, tag_1))
C ← 0
t_start ← Now()
t_end ← t_start + T
while Now() < t_end do
    y ← G(L)
    k ← Extract(s, y)
    r ← Expand(k, tag_2, n)
    C ← C + 1
end while
return C
```

When run on a Macbook Pro laptop with a 3.3 GHz Intel Core i7 CPU and 16GB of RAM, we obtain the numbers in Table I. Thus, the wrapper is about four times slower than directly accessing the raw randomness. This is more caused

Algorithm		Loop iterations	Throughput
1	Baseline	1053668	16.9 MB/s
2	Wrapper	271000	4.3 MB/s

TABLE I: Throughput based on looping for 1.0s.

by the extreme efficiency of the raw randomness code than by the speed of Extract/Expand. However, as we show below, this is dominated by other factors in real-world protocols (or, in fact, any other cryptographic operation in general).

B. Wrapper overhead in the context of TLS 1.3

To evaluate the cost of the wrapper in a real-world context, we considered four scenarios: using TLS 1.3 server and client without wrappers, both with wrappers, and the scenarios where only one end uses the wrapper. Using the wrapper implementation in the context of the BoringSSL library, we started a local server and client and ran 10000 connections for each scenario. We averaged the times afterwards. In more detail, the measurements are based on a TLS 1.3 connection, where the symmetric cipher is `TLS_AES_128_GCM_SHA256`, the ECDHE curve is `X25519`, and the signature algorithm is `ecdsa_secp256r1_sha256`. This yields the results in Table II.

Config	Client	Server	Average time (ms)	Percent difference
Baseline	raw	raw	0.01459430	0.00
One-sided	wrapper	raw	0.01470534	0.76
One-sided	raw	wrapper	0.01473091	0.94
Full	wrapper	wrapper	0.01476728	1.19

TABLE II: Wrapper overhead in TLS 1.3: no wrapper, one-sided wrappers, and wrappers on both sides.

The results indicate about 1% overhead in the context of TLS 1.3 connections for running the wrapper on both the client and the server.

VII. Conclusion

In this work we provided a new wrapper construction for random number generators that strictly reduces the security impact of untrusted randomness, which is a feature that is unfortunately still lacking in many modern security protocols. Our construction provides a practical solution for this issue. It depends on the specific use of a long-term signature key, which makes it compatible with existing HSM deployments, and allows for the re-use of existing signature keys without breaking cryptographic separation. Furthermore, because our construction does not change the observable behaviour of any code that uses it, it can be deployed incrementally.

In the context of a protocol like TLS 1.3, this means our construction can be deployed on only some servers or some clients without disrupting operation, and without generating new keys, while immediately providing stronger security guarantees. This makes the construction especially useful for, e.g., servers with HSMs that get deployed in untrusted environments, such as for content delivery networks (CDNs). However, the security provided by our construction has many more obvious benefits, and would have effectively prevented the exploitation of weaknesses such as the Dual EC backdoor and the Debian random number generator vulnerability. In the former case of the Dual EC backdoor, it would have prevented reconstructing the internal state from the observable randomness; and even if it were reconstructed, an attacker would still need the signature of the targeted party. For the latter case of the Debian RNG, the pool of generated numbers was small and shared by all devices. With our wrapper, even though the space of the generated raw random numbers would be too small, the signature would have caused each device to have its own unique pool, meaning that attacks would have become device specific; and the dynamic second tag would increase the output space, thereby increasing the pool size beyond that of the flawed RNG.

We formally defined the security property provided by our construction in terms of a game between a challenger and an adversary and provided a security proof. Our analysis shows that the construction fulfills the claimed security requirements: if the adversary learns only one of the signature or the usual randomness generated on one particular instance, then under the security assumptions on the used primitives, the wrapper construction should output randomness that is indistinguishable from a random string.

Our proof-of-concept implementation shows that the wrapper is feasible and that the computational overhead is negligible with respect to the other operations of a real-world security protocol such as TLS 1.3. Thus, we showed that our construction improves the generation of pseudorandom numbers and provides concrete security guarantees for a generic class of protocols at negligible cost to efficiency.

Acknowledgments The authors would like to thank Christopher Wood for providing implementation experiments.

References

[1] RDRAND on AMD CPUs does not work, 2019. https://github.com/systemd/systemd/issues/11810 (Accessed Oct 9th, 2019).

[2] Paolo Abeni, Luciano Bello, and Maximiliano Bertacchini. Exploiting DSA-1571: How to break PFS in SSL with EDH, July 2008.

[3] Erdem Alkim, Roberto Avanzi, Joppe W. Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. https://newhopecrypto.org/, March 2019. Submission to the NIST Post-Quantum Cryptography standardization project, Round 2.

[4] Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, pages 602–619, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keyed hash functions and message authentication. In *Proceedings of Crypto'96*, pages 1–15. Springer, 1996.

[6] Daniel J Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: A Standardized Back Door. Technical report, Cryptology ePrint Archive, Report 2015/767, 2015.

[7] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A Systematic Analysis of the Juniper Dual EC Incident. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 468–479, New York, NY, USA, 2016. ACM.

[8] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 319–335, San Diego, CA, August 2014. USENIX Association.

[9] Shaanan Cohney, Andrew Kwong, Shachar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR_DRBG. Cryptology ePrint Archive, Report 2019/996, 2019. https://eprint.iacr.org/2019/996.

[10] Shaanan N. Cohney, Matthew D. Green, and Nadia Heninger. Practical state recovery attacks against legacy RNG implementations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 265–280. ACM, 2018.

[11] C. Cremers, L. Garratt, S. Smyshlyaev, N. Sullivan, and C. Wood. Randomness Improvements for Security Protocols – Revision 7, 2018. https://www.ietf.org/id/draft-irtf-cfrg-randomness-improvements-07.txt.

[12] Benjamin Dowling. *Provable security of internet protocols*. PhD thesis, Queensland University of Technology, 2017.

[13] Michèle Feltz and Cas Cremers. Strengthening the security of authenticated key exchange against bad randomness. *Designs, Codes and Cryptography*, pages 1–36, 2017.

[14] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, https://www.ietf.org/rfc/rfc2409.txt.

[15] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 205–220. USENIX Association, 2012.

[16] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996, https://www.ietf.org/rfc/rfc5996.txt.

[17] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, https://tools.ietf.org/html/rfc4301.

[18] Alex Klyubin. Some SecureRandom thoughts, 2013. Available online: https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html (Accessed January 2020).

[19] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF), 2010. https://tools.ietf.org/html/rfc5869.

[20] Hugo Krawczyk. HMQV: A High-Performance Secure Diffie-Hellman Protocol. Cryptology ePrint Archive, Report 2005/176, 2005. https://eprint.iacr.org/2005/176.

[21] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. Cryptology ePrint Archive, Report 2010/264, 2010. https://eprint.iacr.org/2010/264.

[22] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Annual Cryptology Conference*, pages 631–648. Springer, 2010.

[23] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In *Provable Security*, pages 1–16. Springer, 2007.

[24] Thomas Lendacky. x86/CPU/AMD: Clear RDRAND CPUID bit on AMD family 15h/16h, 2019. https://lore.kernel.org/patchwork/patch/1115413/ (Accessed Oct 9th, 2019).

[25] Hin-Tak Leung. rdrand instruction fails after resume on AMD family 22 CPU, 2014. https://bugzilla.kernel.org/show_bug.cgi?id=85911 (Accessed Oct 9th, 2019).

[26] Rob Marvin. Google admits an android crypto PRNG flaw led to Bitcoin heist (August 2013).

[27] Tsutomu Matsumoto, Youichi Takashima, and Hideki Imai. On seeking smart public-key-distribution systems. E69(2).

[28] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. Randomly Failed! The State of Randomness in Current Java Implementations. In *CT-RSA*, volume 7779 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2013.

[29] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, https://tools.ietf.org/html/rfc8446.

[30] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 15–27. ACM, 2009.

[31] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, https://www.ietf.org/rfc/rfc4251.txt.