Appeared in the *3rd IEEE Symposium on Security in Networks and Distributed Systems, 2007, pp. 463–469.*

Random Oracle Instantiation in Distributed Protocols Using Trusted Platform Modules

Vandana Gunupudi
vgunupudi@gmail.com

Stephen R. Tate
srt@cs.unt.edu

Dept. of Computer Science and Engineering
University of North Texas
Denton, TX 76203

Abstract

The random oracle model is an idealized theoretical model that has been successfully used for designing many cryptographic algorithms and protocols. Unfortunately, a series of results has shown that proofs of security in the idealized random oracle model do not translate into security in the standard model (basically synonymous with "real systems"), so the reasoning that protocols designed using random oracles are secure on real systems is heuristic at best, and fundamentally flawed at worst. In this paper, we consider how architectural changes taking place in real systems today, specifically the introduction of the trusted platform module, affect the realizability of random oracles. In particular, we show how a TPM that is only trivially enhanced from real, standard TPMs can leverage one of its most powerful capabilities — the capability of keeping secrets from the host in which it resides — in order to provide functionality that is indistinguishable from a true random oracle to any polynomial time adversary. In addition to a careful description of how this works, we provide security proofs based on assumptions of TPM security, and provide concrete performance estimates through benchmarks using a current TPM. To prove the security of our TPM-based scheme, we formally define and prove properties about a cryptographic primitive which we call a "hybrid pseudorandom function" that may be of independent interest.

1 Introduction

Arguably the most successful theoretical model used to design practical cryptographic algorithms and protocols is the random oracle model [6]. This model has been successfully used to design such cryptographic techniques as the Probabilistic Signature Scheme (PSS) [7] and Optimal Asymmetric Encryption Padding (OAEP) [16], which are widely used in practice. The idea with this model is to prove a cryptographic scheme secure in the random oracle model, where all parties including the adversary have access to a random function, called a *random oracle*, and then replace the random oracle with a "good" cryptographic hash function in the standard model. A proof of security in the random oracle model is then taken as evidence that a scheme is secure in the standard model if the instantiation of the oracle by the hash function is secure, although this reasoning has some weaknesses as we describe below.

One of the proposed instantiations of a random oracle in the real world is a pseudorandom function (PRF). A PRF produces output that is computationally indistinguishable from random to a polynomial-time attacker. PRFs are natural candidates for instantiating random oracles; however, this approach does not give the strong guarantees that one would like, and problems with this were noted even in the original and influential random oracle paper by Bellare and Rogaway [6]:

> The cryptographic primitive suggested and constructed for this purpose [replacing a random oracle] is the pseudorandom function (PRF). For a PRF to retain its properties, however, the seed via which it is specified (and which enables its computation) must remain unknown to the adversary. Thus the applicability of the paradigm is restricted to protocols in which the adversary is denied access to the random oracle.

In protocols in which a party participating in the protocol can be corrupt, and yet must have access to the random oracle, it is impossible in a traditional model of computation to keep the seed from the adversary. This was explored rigorously by various authors [3, 9] who showed that there exist schemes that are secure in the random oracle model but are uninstantiable in the standard model. Bellare *et al.* [3] define a scheme as *uninstantiable* with respect to some goal if there is a secure implementation of the scheme in the

random oracle model that meets this goal but no instantiation of the scheme meets the goal in question in the standard model. When instantiating the scheme in the standard model, the random oracle is replaced by some family of functions. Canetti, Goldreich and Haveli [9, 15] showed that there exist uninstantiable schemes for the cryptographic goals of IND-CPA secure encryption and digital signatures secure against chosen message attacks. Bellare *et al.* [3] showed that the Hash ElGamal scheme, a hybrid encryption scheme, is uninstantiable for the goal of IND-CCA secure asymmetric encryption. While these negative results don't imply that all schemes designed using random oracles are insecure, they do show using security proofs that rely on the full power of random oracles to imply security in the standard model are inherently flawed, and security guarantees in such a situation are at best heuristic. However, this does not mean that all is lost for schemes designed using the random oracle model — it is still quite possible that an algorithm designed using random oracles could require less strict requirements than complete randomness, and hence instantiations that preserve security under weaker requirements could work.

Motivated by the importance of the random oracle model and these impossibility results, in this paper we explore the possibility of utilizing the unique functionality of the trusted platform modules (TPMs) proposed by the Trusted Computing Group [17] to instantiate a random oracle, when all parties have access to a TPM. A TPM has precisely the capability that is missing in the preceding discussion regarding the use of a PRF: the ability to keep secrets from the platform owner while allowing the use of those secrets in carefully controlled ways. Even more importantly, a type of key introduced in the latest version of the TPM specification (version 1.2), called a "Certifiable Migratable Key" (CMK)[1], allows a set of TPMs to establish a shared secret in such a way that all parties have assurance that these secrets have never been available outside of a protected TPM environment. Using such a shared secret as a seed to a PRF would then allow a platform to utilize this PRF with an unknown seed in place of a random oracle. What this does is allows us to move from heuristic arguments based on random oracles to rigorous proofs based on an assumption that TPMs securely implement their specified functionality (in addition to some standard complexity assumptions). While this means we rely on an additional assumption, we feel this is in many instances a better foundation than the demonstrably flawed reasoning regarding random oracles being instantiated in the standard model.

Our **main contributions** include

- a secure instantiation of random oracles in multi-party protocols using a combination of existing and easily provided TPM functionalities;
- careful analysis and rigorous security proofs of our construction;
- a formal definition and a security proof of a new cryptographic primitive called a "hybrid pseudorandom function";
- benchmarks taken using an actual TPM that allows us to analyze the practical efficiency of our technique; and
- the solution of an interesting subproblem, that of using CMKs to establish a shared secret without needing the participation of a migration authority as an active trusted third party.

The rest of this paper is organized as follows: In Section 2 we discuss relevant TPM functionality and proposed extensions to the functionality. For background on trusted platforms, refer to [1]. Section 3 is our main result section, where we describe the random oracle model, present our techniques and prove security properties. In Section 4 we report some benchmarks of TPM operations that we performed, and use those timings in an efficiency analysis of our technique. Finally, Section 5 wraps up the paper with a discussion of remaining problems and summarizes our results.

2 Our Use of TPM Functions and Extensions

In this section, we present an overview of the TPM functionalities that we use in our construction. For a detailed description of TPMs and their features, refer to [1]. Version 1.2 of the TCG specification introduced a new protection level for keys protected by a TPM: **Certifiable Migratable Keys (CMKs)**. A CMK is a migratable key in which the migration is restricted — when the key is generated (inside a TPM), a list of public keys of "migration authorities" is committed to, and any migration of this CMK must be done by encrypting the private portion of the key using one of these previously-specified public keys. Since the authorized public keys are specified when the CMK is created, the CMK is bound to this set of migration authorities for the lifetime of the key, and the TPM can certify (sign) the key along with the migration authority list; therefore, any receiver of this certificate obtains assurance that the key exists only in the original TPM or in places authorized by one of this fixed set of migration authorities. The intended use of CMKs in the TCG specification is that a public, trusted migration authority would publish a Migration Practice Statement (similar to a Certificate Practice Statement for an X.509 Certification Authority), so can provide

[1] Note that in various parts of the TCG specifications, this key type is referred to as "certified migration key" and "certifiable migration key" in addition to "certifiable migratable key."

assurance that CMKs are only migrated to TPM-protected environments. In Section 3.1 we will show how CMK operations can be used to migrate a key to a specific TPM-protected environment without needing a trusted migration authority.

Ideally, we would like to design protocols that work with existing, standard TPMs. However, that is not possible in this case, and we require three modifications to a standard TPM. Two of these modifications are trivial, and the third would be easy to accomplish but is a more substantial change.

The first trivial change is to add the capability for using a TPM secret as a key to HMAC, which we will use as a PRF with an unknown seed. Since TPMs have protected storage for secrets and must support HMAC for other operations, this is simply a matter of adding the right command to the TPM's command set.

The second trivial change is the creation of a new key type, usable only as a secret key for HMAC in the command we just described — consistent with the TCG specification naming, we suggest a new `TPM_KEY_USAGE` value with the name `TPM_KEY_HMAC`. We note that this change is entirely optional, but we feel it is desirable for two reasons: First, as a valid key type sent to the `TPM_CMK_CreateKey` command, we could avoid the costly operation of generating a new asymmetric key, which is completely unnecessary in our situation. Second, good cryptographic practice states that keys should be used for a single purpose — while we could, for example, create a signing key and use this as a shared secret (this is in fact what we do in Section 4 to benchmark our operations on an existing TPM), it unnecessarily complicates security arguments when we have to consider the impact of operations unrelated to our intended use for this key.

Our third TPM modification is more significant, and is necessary due to the following chicken-and-the-egg problem: Our overall goal is to instantiate a random oracle, and make rigorous security arguments which are free of the kinds of problems that arise in instantiating random oracles in the standard model. However, the standard TPM facility for migrating a CMK uses RSA encryption with OAEP, a random-oracle designed scheme that uses a standard hash function (SHA-1 in the TPM) for the random oracle. While such a system *may* be secure, it does not allow us to have a properly self-contained security proof. Fortunately, in order to perform CMK migration, we only need encryption and signature capability, and techniques are known for both of these that are secure in the standard model (not relying on random oracles). For concreteness, we assume that encryption is done using the Cramer-Shoup CCA-secure encryption scheme [11] and signatures are performed using Fischlin's modification [12] to the Cramer-Shoup signature scheme [10]. Both of these schemes have been proven secure without the need for random oracles, with security that is based on standard complexity assumptions (the decisional Diffie-Hellman and strong RSA assumptions) and requiring operations of modular exponentiation, which must be already supported by TPMs due to the use of RSA in a standard TPM.

3 Implement a Random Oracle with a TPM

In the random oracle model, parties are assumed to have access to a random function $H : \{0,1\}^* \rightarrow \{0,1\}^*$ which is universal and consistent across all parties, so that if Alice requests a value $H(x)$ from the oracle and Bob later also requests $H(x)$, they should obtain the same value. However, the function is truly random, so having sampled several values from H should not give any information about the value of H on values which have not been specifically requested from the oracle. In practice, the domain and range of the random oracle are determined by the requirements of the cryptographic scheme using the random oracle, and are typically fixed-length binary strings where the length depends on a security parameter.

Due to the popularity of the random oracle methodology, whereby a lot of encryption and signature schemes were proven secure in this model, efforts [8] were made to identify properties that are required for real-world instantiation of these oracles. Refer to our full paper [13] for a detailed literature survey on the topic. Various candidates for instantiation of random oracles in the real world have been proposed. Various researchers [4, 5] have shown that a pseudorandom function (PRF) is a natural candidate for replacing a random oracle. We propose using a slightly modified TPM to compute the PRF while keeping the seed secret. Specifically, we use Certifiable Migratable Keys (CMKs), which are keys certified by the TPM and which can be used for computations at the request of the host, but are kept secret within the TPM. Since the TPM already possesses an internal HMAC engine for calculating and storing HMAC digests, it is simple enough to add an interface for performing HMAC operations using CMKs, as we described in Section 2. Since HMAC is proven to be a PRF under the assumption that the underlying compression function is a PRF [2, 4], we can use the CMK as the seed for an HMAC-based PRF when all parties have access to TPMs. While there are some concerns regarding whether HMAC is a PRF when using specific hash functions with recently discovered weaknesses, such as MD5 and SHA1 [14], we assume that an appropriate underlying hash function is used.

3.1 Establish a Shared Secret with CMKs

In this section, we describe how standard TPM operations on CMKs can be used to establish a secret that is

shared between the TPMs of a fixed set of parties with known public keys for non-migratable keypairs. As described earlier, when a CMK is created it is bound to a list of public keys of migration authorities (MAs), and any migration of this CMK must be coordinated by one of these migration authorities. In a typical TPM application, the MA would be a public entity, trusted to migrate keys in accordance with a published migration policy. In our case, we avoid having an independent MA by migrating keys "under the authority" of a non-migratable storage key on a destination platform — since this isn't an actual MA key (a key of type `TPM_KEY_MIGRATE`), it cannot be further migrated, so is effectively contained within that particular destination platform.

In the following, we have parties $A_1, A_2, \cdots, A_n$. Each party A_i has a TPM T_i, a properly certified identity key I_i, and a non-migratable storage key P_i which will be the parent of the shared secret in A_i's protected storage hierarchy. The shared secret will be generated internally to T_1, and then transferred (migrated) securely to TPMs $T_2, T_3, \cdots, T_n$. We break down the shared secret establishment into three phases, destination certification, secret creation/migration, and destination secret installation, which we describe below. We omit the security proof of our construction which is available in the full version [13] of our paper.

Destination Certification. For each $i \in \{2, \cdots, n\}$, A_i uses the TPM command `TPM_CertifyKey` to create a certification for key P_i using identity key I_i. Note that this can be done in advance — any time after P_i and I_i have been created.

Secret Creation/Migration. A_1 collects all of the public keys corresponding to $P_2, \cdots, P_n$, along with their certifications and corresponding identity keys, and verifies that these are all certified as non-migratable storage keys using an identity key that is in turn certified by a trusted PrivacyCA. This list of public keys is then used as the list of authorized migration authorities when the CMK K is created using `TPM_CMK_CreateKey`. A_1 uses `TPM_CertifyKey2` to create a certification for key K using identity key I_1. To transfer K, A_1 uses `TPM_CMK_CreateBlob` for each destination key $P_2, \cdots, P_n$ to create migration blobs (re-encrypted private keys[2]) for each destination, and transmits the appropriate blob along with a copy of K's certification and the list of certified parent keys $P_2, \cdots, P_n$ to each of $A_2, \cdots, A_n$.

[2]Note that actual migration is slightly more complex, with each migration blob having an associated "random part" which is useful in certain scenarios — in our situation, we simply treat the migration blob and random part as a single unit of data, and transmit them together.

Destination Secret Installation. When A_i (for $i \in \{2, \cdots, n\}$) receives the migration blob and K's certification, it uses `TPM_ConvertMigrationBlob` which installs K under parent key P_i in A_i's storage hierarchy. Next, A_i verifies that key K is a CMK certified by an identity key which is in turn certified by a trusted PrivacyCA, and that its migration is restricted to keys $P_2, \cdots, P_n$, each of which is also verified as a non-migratable storage key certified by an identity key that is certified by a trusted PrivacyCA.

Theorem 3.1 *If $T_1, \cdots, T_n$ are properly functioning, non-compromised TPMs, and if the PrivacyCAs certified only legitimate TPM-bound identity keys, then at the end of this protocol each TPM has a copy of K which is internally usable and not available to other TPMs or outside of the TPM-protected environment.*

3.2 A TPM-Oracle

In the previous section, we described how parties equipped with TPMs can establish a shared secret in such a way that all parties have assurance that the secret is only available to a fixed list of parties and only within a TPM-protected environment. In this section, we will describe how to use this shared secret within a TPM to create what we refer to as a "TPM-Oracle" — a computation done inside the TPM that is polynomial-time indistinguishable from a random oracle. The following definitions formalize several important concepts.

Definition 3.1 *Let $l(k)$ and $m(k)$ be polynomially bounded length functions, where k is interpreted as a security parameter, and let $\mathcal{X}_k(x)$ for $k \in Z$ and $x \in \{0,1\}^{l(k)}$ be a collection of random variables (an "ensemble") drawn from $\{0,1\}^{m(k)}$. We write $\mathcal{X}_k$ to denote the collection of random variables $\mathcal{X}_k(x)$ with k fixed, and write $\mathcal{D}(\mathcal{X}_k)$ to denote an algorithm $\mathcal{D}$ that can sample random variables $\mathcal{X}_k(x)$ for any $x \in \{0,1\}^{l(k)}$. A distinguisher for ensembles $\mathcal{X}_k$ and $\mathcal{Y}_k$ is an algorithm D with boolean output such that for all sufficiently large k,*

$$|Prob[\mathcal{D}(\mathcal{X}_k) = 1] - Prob[\mathcal{D}(\mathcal{Y}_k) = 1]| > \frac{1}{p(k)} ,$$

where $p(k)$ is some polynomial in k. We say ensembles $\mathcal{X}_k$ and $\mathcal{Y}_k$ are computationally indistinguishable if there exists no probabilistic polynomial time distinguisher for these ensembles.

Definition 3.2 *Consider a family of functions $PR_s : \{0,1\}^{l(k)} \rightarrow \{0,1\}^{m(k)}$ that is indexed by a variable $s \in \{0,1\}^k$, which we call the seed. For this family of functions, if we select s uniformly from $\{0,1\}^k$ then this defines an ensemble $PR_k(x)$ in a natural way. We also consider the*

ensemble where each $R_k(x)$ is uniformly distributed over $\{0,1\}^{m(k)}$. Function family PR_s is a pseudorandom function family (PRF) if PR_k is computationally indistinguishable from R_k.

The desired random-oracle query is a call to a function $H : \{0,1\}^{a(k)} \rightarrow \{0,1\}^{b(k)}$, where k is a security parameter, and $a(k)$ and $b(k)$ are polynomially-bounded functions — note that some random-oracle-designed protocols require multiple random oracles with different domain and range sizes, and in this paper we consider these as independent oracles with separate and independent shared secrets, but each with its own well-defined domain and range size. Our TPM-Oracle is denoted $\mathcal{TO} : \{0,1\}^{a(k)} \rightarrow \{0,1\}^{b(k)}$.

We assume that our HMAC engine uses a fixed Merkle-Damgård-style hash function that has an underlying compression function that uses $c(k)$-bit blocks and produces $d(k)$-bit digests, defining a function $HMAC : \{0,1\}^{c(k)} \times \{0,1\}^* \rightarrow \{0,1\}^{d(k)}$, and we use notation $HMAC(s,m)$ to denote evaluating HMAC using secret s and message m. Recall that our shared secret, established using CMK operations, is $K \in \{0,1\}^{c(k)}$, and we use a TPM-Oracle to answer queries of the form $H(x)$ for some party A_i. Finally, let $bin(b,x)$ denote the b-bit binary representation of x (assuming $x < 2^b$). Our TPM-Oracle is then defined as follows:

$\mathcal{TO}(x)$
  $m \leftarrow \left\lceil \frac{b(k)}{d(k)} \right\rceil$
  $b \leftarrow \lceil \log_2(m) \rceil$
  $A \leftarrow HMAC(K, bin(b,0)||x)|| \, HMAC(K, bin(b,1)||x)$
    $|| \cdots || \, HMAC(K, bin(b, m-1)||x)$
    return The first $b(k)$ bits of A

$\mathcal{TO}$ is a standard secure extension of a PRF, and is itself a secure PRF if the underlying building block is a secure PRF. Bellare has shown that HMAC is a PRF if the underlying compression function is a PRF [2], so we get the following lemma regarding $\mathcal{TO}$.

Lemma 3.1 *If the compression function used by HMAC is a pseudorandom function, then $\mathcal{TO}$ is a pseudorandom function.*

3.3 Security of the TPM-Oracle

We now consider the security of our complete system, including the secret establishment and use of the TPM-oracle, and provide a proof to show how this system provides security guarantees equivalent to a true random oracle with respect to a polynomial-time adversary.

We define a new construction called a **Hybrid PRF** system which comprises two parts. First a Key Encapsulation Mechanism (KEM) is used to encrypt a random key. Then the key is used as input to a Pseudorandom Function (PRF) to produce pseudorandom output. We first present definitions of Key Encapsulation Mechanism as given by Cramer and Shoup [11] and then expand this to a description of our Hybrid PRF functionality.

Definition 3.3 ***Key Encapsulation Mechanism (KEM)*** *consists of the following polynomial time algorithms:*

- *Key Generation Algorithm: A probabilistic algorithm that generates public and private keys (pk, sk) based on security parameter λ. We also define a key space $\mathcal{K}_K$ which is determined by λ, and is typically the set of binary strings of a length which can be encrypted under pk.*

  $(pk, sk) \leftarrow KEM.Gen(1^\lambda)$

- *Encryption Algorithm: A probabilistic algorithm that generates $K \in \mathcal{K}_K$ and the encryption ϕ, of K.*

  $(K, \phi) \leftarrow KEM.Enc_{pk}()$

- *Decryption Algorithm: A deterministic algorithm that decrypts a ciphertext ϕ and returns either K (if the decryption succeeds) or $\perp$ (if it doesn't).*

  $K \leftarrow KEM.DEC_{sk}(\phi)$

Definition 3.4 *A **Hybrid-PRF** consists of the following:*

1. *Key Generation Algorithm: A probabilistic algorithm that generates public and private keys (pk, sk). The public key defines the KEM key space $\mathcal{K}_K$.*

   $(pk, sk) \leftarrow HPRF.Gen(1^\lambda)$

2. *Encryption Algorithm: A probabilistic algorithm, that given pk, generates and encrypts a random key $K \in \mathcal{K}_K$ with the corresponding secret key, sk and returns ϕ, the encrypted key.*

   $(K, \phi) \leftarrow HPRF.Enc_{pk}()$

3. *Decryption Algorithm: A deterministic algorithm that returns K as the decryption of ϕ.*

   $K \leftarrow HPRF.Dec_{sk}(\phi)$, *output $\perp$ if $K \notin \mathcal{K}_K$*

4. *PRF algorithm: A deterministic algorithm, that given a key K and a message, m, computes a hash of the message using the key as input to a keyed PRF.*

   $r \leftarrow HPRF.F_K(m)$

Security of schemes for these functionalities is defined in terms of games that an adversary plays against the scheme, and we use Adv_{KEM}, Adv_{PRF}, and Adv_{HPRF} to denote the advantage of the best adversary in games for KEM, PRF,

and HPRF, respectively. These systems are secure if the adversary's advantage is negligible. These games and definitions are standard and can be found in many places in the cryptographic literature, or an interested reader can find definitions and discussion in the full version of this paper [13].

Theorem 3.2 *If a given PRF is secure against the standard PRF game and the KEM used to encrypt the key K is CCA-secure, then the hybrid PRF construction is secure. In particular, $Adv_{HPRF} \leq 2Adv_{KEM} + Adv_{PRF}$.*

The proof of this theorem can be found in our full paper [13]. In summary, from Theorem 3.1, Theorem 3.2 and Lemma 3.1, we obtain the following result.

Corollary 3.1 *Assume that we have properly functioning, secure TPMs that use the above technique for creating a TPM Oracle. If the hash function used by HMAC has a pseudorandom comrpession function, and the CMK transfer scheme uses a CCA-secure public key cryptosystem, then any polynomial time algorithm secure in the random oracle model is secure in the TPM oracle model.*

4 Benchmarks and Performance Analysis

In this section, we report some benchmarks on basic operations performed by an existing TPM, and use these benchmarks to estimate the performance of our protocol. We must extrapolate from basic benchmarks due to our protocol's requirement of currently unavailable functionality (specifically, the use of HMAC with an internal TPM secret key). For our tests, we used a desktop machine with an Intel D945GTP-LKR motherboard, which incorporates an version 1.2 TPM made by STMicroelectronics (model ST19WP18). We ran each of our timed operations at least 4 times, and give the average time for each operation in the following table.

Operation	Time (sec)
`TPM_LoadKey2`	3.03
`TPM_CreateKey`	33.40
`TPM_Seal`	0.39
`TPM_Unseal`	1.19
`TPM_CertifyKey`	1.39

Only the `TPM_CreateKey` showed any significant variability in the time between different tests, which is understandable due to the probabilistic nature of generating RSA keys. While creating a new asymmetric key is currently the only way to establish a CMK and hence a shared secret, which is why we include that time above, our protocol does not require an asymmetric key as a CMK — as we discussed in Section 2 we suggest the use of a simple randomly generated secret, in which case the time for creating such a CMK would be dominated by the time to wrap (encrypt) this secret using the parent key. To estimate the time required for this, we benchmarked the `TPM_Seal` operation, which does essentially the same operations, and we use this time as an estimate of how long a CreateKey operation would take with our new type of key. We assume that verification of certificates by the host platform takes insignificant time compared to these TPM operations — RSA keys used by the TPM use a fixed public exponent of 65,537, and on a modern PC-class processor such a verification can be done in less than a millisecond.

Performance of secret creation/migration: As described above, we estimate the time of creating our special CMK key using the `TPM_Seal` command, which takes 0.39 seconds, and certifying this key requires loading both the CMK and the appropriate AIK (3.03 seconds each) and then doing the certification (1.39 seconds), for a total creation and certification time of 7.84 seconds. A migration blob is created for each of the n destinations, taking a total of $3.03(n-1)$ seconds. Combining these results, the total time for the secret creation/migration phase is

$$4.81 + 3.03n \text{ seconds.}$$

In particular, for a two-party protocol, the estimated time is 10.87 seconds.

Performance of destination secret installation: On each destination receiving the CMK secret, the destination TPM must do a blob conversion, taking 1.58 seconds (an unseal followed by seal). The remaining operations are certificate verifications, which are done by the host processor rather than the TPM, so take insignificant time compared to the 1.58 seconds for the blob conversion.

Performance of TPM-Oracle queries: There is no current user interface to the HMAC engine within a TPM, but there is a direct way to use the SHA1 engine. We tested SHA1 on inputs of size 16k, 32k, 48k, and 64k, and found a very consistent increase of 1.935 seconds per 16k increment. Based on this, we estimate that one application of the internal SHA1 compression function takes no more than 7.6 milliseconds — note that the since the hashed data had to be passed into the TPM over a relatively slow interface, this timing may be as much constrained by platform-to-TPM communication speed as it is by the computational speed of the SHA1 engine, which would not be as much an issue with our usage. Based on these benchmarks, we see that if our TPM-oracle is answering queries from a domain that requires b_i input blocks to hash, and we are generating an output that needs b_o output blocks, then the time would be at most $7.6(b_i + 3)b_o$ ms.

To make this concrete, note that if we are creating a TPM-oracle that takes 1024-bit inputs (so $b_i = 3$) and produces 160-bit outputs (so $b_o = 1$), each random oracle

query would take no more than 45.6 milliseconds.

In summary, our TPM-oracle using current hardware takes around 10 seconds of setup time, and less than 100 milliseconds for most realistically sized TPM-oracle queries. We feel that this is quite practical, and future improvements to TPMs which reduce the setup time would only make it more clearly practical.

5 Conclusion and Open Problems

In this paper, we have shown how a random oracle can be instantiated in a multi-party setting if each party has access to a trusted platform module (TPM). Utilizing a special kind of key called Certifiable Migratable Key (CMK) that can be certified by a TPM and migrated to other TPMs, we can instantiate a random oracle using HMAC-based pseudorandom functions. Our method requires a few modifications to the TPM specification, but none of the modifications are particularly difficult or unreasonable. We provided rigorous proofs that under an assumption of secure TPMs (and standard complexity assumptions) our construction is computationally indistinguishable from a random oracle to a polynomial-time attacker. In the process, we formally define and prove properties about a new cryptographic primitive which we call a "hybrid pseudorandom function" that may be of independent interest.

An interesting open problem is to improve upon the secret establishment process. Currently, if we wish to avoid having an active trusted migration authority, then public keys for all parties must be known at the beginning of the computation, which is particularly difficult in the case of non-interactive zero-knowledge proofs where we don't know who the verifier may be. This is also a problem for dynamic multi-party settings, where parties may join or leave the group during the execution of the protocol. Unfortunately, this seems to be a very difficult problem if the parties are mutually untrusting — there is no obvious way to have a key that can migrate multiple hops without a transitive trust relationship, which is not directly available with the current TPM design. An additional problem related to secret establishment is that, even in a static case where all keys are known, the secret distribution scheme does not scale well as the originating party must send the packaged secret to every participant, taking $\Theta(n)$ time. A tree-structured distribution scheme with depth $O(\log n)$ would be significantly better, but the same problem with transitive trust and forwarding migrating keys is encountered.

References

[1] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, Upper Saddle River, NJ, 2003.

[2] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *CRYPTO*, pages 602–619, 2006.

[3] M. Bellare, A. Boldyreva, and A. Palacio. An uninstantiable random-oracle-model scheme for a hybrid-encryption problem. In *EUROCRYPT*, pages 171–188, 2004.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, pages 1–15, 1996.

[5] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *Proceedings of the 37th Symposium on Foundations of Computer Science (FOCS)*, pages 514–523, 1996.

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

[7] M. Bellare and P. Rogaway. The exact security of digital signatures - How to sign with RSA and Rabin. In *EUROCRYPT*, pages 399–416, 1996.

[8] R. Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *CRYPTO*, pages 455–469, 1997.

[9] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[10] R. Cramer and V. Shoup. Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security*, 3(3):161–185, 2000.

[11] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 33(1):167–226, 2003.

[12] M. Fischlin. The Cramer-Shoup Strong-RSA signature scheme revisited. In *Public-Key Cryptography (PKC)*, pages 116–129, 2003.

[13] V. Gunupudi and S. R. Tate. Random oracle instantiation in distribued protocols using trusted platform modules. Technical report, University of North Texas, USA, 2007. Available at http://cops.csci.unt.edu/publications/.

[14] J. Kim, A. Biryukov, B. Preneel, and S. Hong. On the security of HMAC and NMAC based on HAVAL, MD4, MD5, SHA-0 and SHA-1. In *Proceedings of the 5th Int'l Conference on Security in Communications Networks (SCN)*, 2006.

[15] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *Theory of Cryptography (TCC)*, pages 21–39, 2004.

[16] V. Shoup. OAEP reconsidered. In *CRYPTO*, pages 239–259, 2001.

[17] Trusted Computing Group. Website. http://www.trustedcomputinggroup.org.