Analysis of The Porticor Homomorphic Key Management Protocol *

Alon Rosen†

October 18, 2012

Abstract

This document contains a description of the Porticor Homomorphic Key Management (HKM) protocol, along with an analysis of its security under explicit and well defined assumptions.

1 Introduction

The Porticor *Homomorphic Key Management* (HKM) protocol involves the following three entities:

- The Porticor *Virtual Key Management* (PVKM) service (runs on the Porticor server).
- The Porticor *Appliance* (instantiated by the user and remotely running on the cloud).
- The *User* (a human who is represented by a distinct client platform, e.g. a Web browser).

The User holds a *master key*, which is used jointly with the appliance and PVKM key-shares to reconstruct *specific keys* for data encryption.

The role of the PVKM is to assist the Appliance in key management without knowing master keys or specific keys, and enable different instances of the Appliance to consistently reconstruct specific keys by combining corresponding key shares. The specific keys are then used to encrypt various storage objects such as individual files. The protocol's objectives are:

- Preserve secrecy and authenticity of the master and specific keys with respect to the PVKM.
- Preserve secrecy and authenticity of the master and specific keys with respect to an eavesdropper.
- Preserve secrecy and authenticity of the master key with respect to all appliance instances.

We assume that the PVKM is "passive" (aka "semi-honest"), in the sense that it follows the prescribed protocol (see Section 4). The eavesdropper may be "active", which means that beyond eavesdropping it could also try to modify and/or omit messages. Security with respect to the PVKM and appliances is achieved by protocol design, and security against the active eavesdropper is achieved using standardized tools such as SSL (for this we will assume that a trusted Public-Key infrastructure is set-up and available).

The main objective of this document is to provide a mathematical description of the various modes of operation of the HKM protocol, along with corresponding proofs of security. The proofs will be established based on clearly stated and well-defined assumptions on the set-up of the system (such as the extent to which the cloud provider has access to the data used by the appliance, and the PVKM being "passive") and on the security of the underlying cryptographic primitives (such as ElGamal encryption and SSL).

Needless to say that these assumptions are necessary to the security analysis and its validity.

†Efi Arazi School of Computer Science, IDC Herzliya. E-mail: alon.rosen@idc.ac.il

2 Overall Assessment

The HKM protocol has a minimalistic and simple design. This has several advantages:

- It reduces the likelihood of implementation mistakes.
- It allows clean modeling of security and makes the protocol amenable to analysis.
- It opens up the possibility for enhancing the protocol with stronger security guarantees.

The security of the HKM protocol relies on the semantic security of ElGamal encryption with key size $k = 2048$, which is a standard and relatively safe assumption. Functionality of the protocol builds on the (multiplicative) homomorphic properties of ElGamal encryption. These properties hold unconditionally, and enable to manage and derive keys in a consistent manner. Authenticity and secrecy of communication is obtained using SSL, which means that a trusted PKI has to be in place. Again, the assumption that SSL is secure is standard and relatively safe (provided that the implementation is proper).

Overall, I find the protocol design to be sound. In particular, it is possible to state a set of explicit assumptions under which one can argue the protocol implements the intended functionality in a secure manner. Three concrete issues that I identified, and that in my opinion require consideration are:

1. **The protocol theoretically allows the PVKM to infer relations between specific keys.** This issue is predominantly theoretical. In practice it is mitigated in an ad-hoc manner by applying a ”random oracle” to derive the actual keys. This presumably destroys any “sensible” relation between specific keys and makes them “wild” enough to thwart any conceivable related key attack. We elaborate on this issue in Section 4.2 and propose a concrete suggestion on how to further strengthen it (using t-wise independent blinding) in Section 4.3.

2. **The protocol's security holds in the semi-honest model.** This concerns the adversarial model. The assumption that the providers are semi-honest is not unreasonable. It essentially amounts to:

   - Trusting Porticor to have implemented the protocol as specified.
   - Trusting the platform on which the implementation is executed to actually run it as implemented.

   Achieving security in the semi-honest model is already a non-trivial task and can itself be considered as a significant milestone.

   However, it will not cover the user in case that any one of the above assumptions does not hold: by arbitrarily deviating from the prescribed instructions it may actually be possible to learn sensitive information such as the master key.

   This can be further alleviated, since the simple design of the HKM protocol enables its compilation so that is secure even in the malicious model. This can be done with quite reasonable performance overhead, using the same cryptographic primitives that are used in the current version of the HKM.

3. **Specific keys reside in an appliance instance's memory on the clear.** This has implications on the exposure of specific keys to the environment in which the Appliance is running. It causes the security of encrypted data to implicitly rely on the assumptions that the Appliance does not leak specific keys, and that the environment in which the appliance resides does not actively attempt to get hold of the keys during the lifespan of the appliance's instances.

   The risk of specific key exposure is diminished by the fact that the appliance is relatively short lived and ephemeral. In addition, the protocols are designed so that the “master key” is never leaked to the appliance. So the damage of having a specific key leaked is limited only to the data encrypted using this specific key.

3 Preliminaries

3.1 ElGamal Encryption

The ElGamal encryption scheme works as follows. The key generation algorithm $\text{Gen}(1^k)$ generates secret key $sk = x$ and public key $pk = (G, g, h = g^x)$, where g is a randomly chosen generator of a group G of prime order q with $|q| = k$ and x is randomly chosen in $\mathbb{Z}_q$.

The plaintext space is G, the randomness space is $\mathbb{Z}_q$ and the ciphertext space is G. We require G to be a group where it is easy to determine membership and compute the binary operations and assume parties check that ciphertexts are valid, by verifying that the ciphertext is an element in G.

To encrypt a value $m \in G$ the sender picks randomness $r \leftarrow \mathbb{Z}_q$ and computes the ciphertext

$$(c, d) = (g^r, h^r \cdot m).$$

We let $\text{Enc}_{pk}(m; r)$ denote the encryption of a value $m \in G$ using randomness r. Note that the ciphertext uniquely determines the value of m. This is because the mapping from $r \in \mathbb{Z}_q$ to the values g^r and h^r is injective. To decrypt a ciphertext (c, d) the receiver uses $sk = x$ in order to compute

$$d/c^x = ((g^x)^r \cdot m)/(g^r)^x = m.$$

The security of the ElGamal encryption scheme is equivalent to the Decisional Diffie-Hellman assumption in G, which states that it is infeasible to distinguish triplets of the form (g^a, g^b, g^{ab}) from triplets of the form g^a, g^b, g^c, where a, b and c are uniformly chosen in $\mathbb{Z}_q$. Under this assumption it can be shown that for any message $m \in G$, the distribution $\text{Enc}(m; r)$ of encryptions of m (over a random choice of r) cannot be feasibly distinguished from the distribution of encryptions of a uniformly chosen message $m' \in G$.

The ElGamal variant described above is multiplicatively homomorphic. For all $m, m' \in G$ and $r, r' \in \mathbb{Z}_q$ we have

$$\text{Enc}_{pk}(m; r) \odot \text{Enc}_{pk}(m'; r') = \text{Enc}_{pk}(m \cdot m'; r + r'),$$

where for $c, d, c', d' \in G$, the value $(c, d) \odot (c', d')$ is defined as $(c \cdot c', d \cdot d')$.

3.2 Concrete Parameters

The group in use for the ElGamal encryption is $G = QR_p$, the group of quadratic residues mod p, where the prime p and a corresponding generator g are sampled once and for all. The prime p is sampled under the condition that it is of the form $2q + 1$ for a random prime number $q \in \mathbb{N}$ of size $k = 2048$. The sampling proceeds by randomly picking numbers q_i until conditions

$$\text{“}q_i \text{ is prime” and “} p_i = 2q_i + 1 \text{ is prime”}$$

are both satisfied. A generator $g \in \mathbb{Z}_p^*$ of order q is obtained by sampling a random number in $\mathbb{Z}_p^*$, squaring it, and testing that the result is not 1. Since q is prime and $p = 2q + 1$, this guarantees that the order of g is q and that g generates $G = QR_p$ (note that the plaintext space is also QR_p).

Since p and g are fixed, whenever it is clear from the context we will refrain from explicitly specifying them as part of a public key of the ElGamal encryption. Thus, to specify a public key pk it will be sufficient to provide a single value $h = g^x \in QR_p$. In such a case, the corresponding public-key is $pk = (G, g, h)$.

In accordance with the HKDF RFC 5869 key-derivation standard [1], specific keys are derived from elements $S_j \in G$ by computing $\text{HKDF}(S_j)$, which yields a 256-bit pseudorandom string (2048 bits of entropy should be more than sufficient for pseudorandomness) to be used as a key for AES-256 encryption.

4 The HKM Protocols

In this section we describe the main protocols of the Porticor HKM system. The protocols are:

1. New Appliance Instance.
2. New Storage Object.
3. Set Protected Item.

The main protocol, called *New Appliance Instance*, involves setting up an instance of the appliance, with the assistance of the user, which holds a "master key" and the PVKM module, which aids in maintaining state (and specifically the various encryption keys) across different instances.

Another important protocol, called *New Storage Object*, enables the creation of a fresh specific key to be used to encrypt the data on the storage device, and allows the PVKM to store a "blinded" version of this specific key for future use. A related protocol, called *Set Protected Item*, enables the appliance to generate specific keys "on the fly" and to transmit a "blinded" version to the PVKM.

We let $n \in \mathbb{N}$ denote the total number of Appliance instances, and let $m \in \mathbb{N}$ denote the total number of specific keys. The security parameter is denoted $k \in \mathbb{N}$. We assume that the traffic between the parties is encrypted and authenticated using SSL. We assume that the user possesses a "master" key, $K \in G$, which will be used throughout the life of the system (in particular across different instances) in order to generate and then reconstruct specific keys. We use capital letters (such as K and S_i) to denote keys, and letters from the Greek alphabet (such as α_i, β_i and γ_i) to denote "masks" that are used for blinding the keys.

4.1 New Appliance Instance

To initiate a new appliance instance the system uses a three-party protocol between the user, denoted U, the i^{th} instance of the Appliance, denoted A_i, and the *PVKM* server, denoted P. The protocol does not require communication between U and P. Its security crucially relies on the following three assumptions:

1. No two of the parties collude.
2. All parties follow the protocol's prescribed instructions.
3. The underlying cryptographic primitives (SSL, ElGamal) are secure (in a well-defined sense).

The user U initiates an appliance instance A_i. The appliance is ephemeral and resides in a compute cloud (i.e. a cloud used for computing), such as an Infrastructure-as-a-Service (IAAS) environment. It can be used to interact with the PVKM P in order to reconstruct specific keys, $S_j \in G$, even if these keys were generated by previous instances of the appliance. The way in which state is kept between different instances of the appliance is by having U consistently use the same master key K in all instances A_i, and by having P store a "masked" version of the specific keys $S_j \cdot K$ (to aid preserving secrecy against the PVKM).

We start by giving a high-level description of the protocol for generating a new appliance instance (note that parties' outputs are local and are assumed to be securely stored by them):

- $P \to A_i$: Send an ElGamal public-key $pk_i = (G, g, h_i)$. Locally store the associated secret key sk_i.
- $A_i \to U$: Forward pk_i.
- $U \to A_i$: Send master key K encrypted under pk_i.
- $A_i \to P$: Homomorphically generate encryption of $K \cdot \gamma_i$, where γ_i is a random "mask".
- P : Use sk_i to decrypt. Output $K \cdot \gamma_i$.
- A_i : Output γ_i.

Intuitively, the secrecy of the master key K with respect to A_i is guaranteed by the assumed security of the ElGamal encryption scheme (note that sk_i is not known to A_i). The secrecy of K with respect to P is guaranteed by the fact that the "mask" γ_i is chosen uniformly at random and is not known to P (the only values that P learns are $K \cdot \gamma_i$).

Protocol (P, A_i, U)

Common input: A pair (G, g), where $G = QR_p$ and g is a generator of G.

Auxiliary input to U: Master key $K \in G$.

$P \rightarrow A_i$: Sample $sk_i \leftarrow \mathbb{Z}_q$ and send $h_i = g^{sk_i}$

$A_i \rightarrow U$: Send $h_i = g^{sk_i}$ (so $pk_i = (G, g, h_i)$)

$U \rightarrow A_i$: Send $(a_i, b_i) = \mathrm{Enc}_{pk_i}(K; r_i)$, where $r_i \leftarrow \mathbb{Z}_q$

$A_i \rightarrow P$: Sample $\gamma_i \leftarrow G$:

- Compute $(c_i, d_i) = \mathrm{Enc}_{pk_i}(\gamma_i; s_i)$, where $s_i \leftarrow \mathbb{Z}_q$
- Send $(C_i, D_i) = (a_i, b_i) \odot (c_i, d_i) = \mathrm{Enc}_{pk_i}(K \cdot \gamma_i; (r_i + s_i))$

P : Output $K \cdot \gamma_i$ (obtained by decrypting (C_i, D_i) using sk_i).

A_i : Output γ_i

Figure 4.1: New Appliance instance A_i.

4.2 New Storage Object

Once an "instance-specific" masking $K \cdot \gamma_i$ of the master key K has been set up it can be used to protect specific keys, S_j, for encrypting storage objects. This is done by having A_i interact with the $PVKM$ (which will store "blinded" versions of the various specific keys). The main challenge here is to guarantee that the j^{th} specific key S_j could also be accessed by subsequent (and totally unrelated) instances of the appliance.

Protocol (P, A_i)

Auxiliary input to P: $K \cdot \gamma_i$

Auxiliary input to A_i: γ_i

$A_i \rightarrow P$: Request to generate j^{th} specific key

$P \rightarrow A_i$: Sample $\alpha_j \leftarrow G$ and send $\beta_{ij} = \alpha_j / (K \cdot \gamma_i)$

A_i : Output $S_j = \beta_{ij} \cdot \gamma_i = \alpha_j / K$

P : Output α_j

Figure 4.2: New Storage Object - Generating specific key S_j via P.

The string S_j output by A_i is by definition the j^{th} specific key S_j. Since the value α_j that is locally stored by P equals $K \cdot S_j$, and since K is secret with respect to P (it is only known by U), we presumably have that S_j is secret with respect to P.

However, it should be noted that this secrecy is not perfect, as P will eventually hold an entire sequence $K \cdot S_1, K \cdot S_2, \ldots, K \cdot S_m$ of specific keys that are "blinded" using the same master key K. This in particular means that P can potentially infer various relations between the keys (such as the ratio $S_j/S_{j'}$ for $j \neq j'$), and opens the door for related key attacks. This is in practice mitigated by applying a 'random oracle' to the keys. Specifically, the actual data-encryption keys are derived from S_j's using the HKDF function [1], which is assumed to behave as a "random oracle".

4.3 t-wise Independent Blinding

One can further strengthen related key attacks by having the master key be a degree-2 polynomial $K(x) = k_2 \cdot x^2 + k_1 \cdot x + k_0$. Then, the specific keys S_j will be blinded with $g^{K(j)}$. The key observation is that it is possible to homomorphically evaluate the encryption of $g^{K(j)}$ given j and encryptions of g^{k_0}, g^{k_1} and g^{k_2}. This is because:

$$\begin{aligned} g^{K(j)} &= g^{k_2 \cdot j^2 + k_1 \cdot j + k_0} \\ &= (g^{k_2})^{j^2} \cdot (g^{k_1})^{j} \cdot g^{k_0}. \end{aligned}$$

The benefit of blinding with a degree 2 $K(x)$ is that the distribution $K(1), \ldots, K(m)$ is pairwise independent. This means that P will not be able to infer relations amongst any pair of $S_j, S_{j'}$ and so the complexity of him mounting a related key attack will be significantly increased (since he could only rely on relations between three and more keys). A straightforward generalization of the above idea is to take $K(x)$ to be of degree t. This will result in t-wise independent blinding, thus mitigating relations between any t or less keys.

4.4 Set Protected Item

In some cases, the appliance A_i will want to generate a specific key, S_j, "on the fly". After having generated the key the appliance will transmit it ("blinded" with the master key K) to the PVKM for future use. The protocol proceeds as follows.

Protocol (P, A_i)

Auxiliary input to P**:** $K \cdot \gamma_i$

Auxiliary input to A_i**:** γ_i

$A_i \rightarrow P$: Sample $S_j \leftarrow G$. Send $\beta_{ij} = S_j/\gamma_i$

P : Output $\alpha_j = \beta_{ij} \cdot (K \cdot \gamma_i)$

A_i : Output S_j

Figure 4.3: Set Protected Item - Generating specific key S_j via A_i.

As before, the string S_j output by A_i is by definition the j^{th} specific key S_j. Since the value α_j that is locally stored by P equals $K \cdot S_j$, and since K is secret with respect to P (it is only known by U), we presumably have that S_j is secret with respect to P. However, as in the previous case, secrecy is not perfect due to the various relations between the keys.

5 Analysis

5.1 Security in the Semi-Honest Model

We consider three party protocols (A, B, C). A protocol consists of three parties that exchange messages in alternating turns. The messages are determined by the parties' (private) inputs and coin-tosses, as well as on the previous messages that they have received. We will say that a protocol (A, B, C) computes a (possibly randomized) functionality

$$f(a, b, c) = (f_A(a, b, c), f_B(a, b, c), f_C(a, b, c))$$

if whenever A, B and C follow the protocol's instructions, on inputs a, b and c respectively, then their respective local outputs equal $f(a, b, c)$. We let λ denote the "empty string". This is used to indicate that a party has no input and/or no output to/in the functionality.

A party's *view* of a protocol's execution consists of its input, coin-tosses, and received messages. We let $\mathsf{VIEW}_A^{(A,B,C)}(a, b, c, 1^k)$ denote a random variable that is distributed according to A's view when the parties' inputs are a, b, c respectively, the security parameter is k, and the coin tosses of all parties are chosen uniformly at random. B's and C's view are symmetrically defined. Since point-wise communication is done via SSL, we can assume that a party's view only consists of messages that were intended to that party.

One could think of the view of a party as the "information" that this party is obtaining from the protocol's execution. In the semi-honest model, any of the parties may apply an arbitrary polynomial time program to the view in order to try and extract secret information from the interaction.

Our definition of security will require that whatever distribution can be (efficiently) sampled from a party's view could be essentially sampled from the input and output available to that party. This is formalized by exhibiting the existence of a probabilistic polynomial time machine (*simulator*) that, given only the input and output available to the party, is able to produce views that are essentially identical to that party's views in actual protocol executions.

Definition 5.1. Let $f(a, b, c)$ be a (possibly randomized) three-input functionality. A three-party protocol (A, B, C) for computing f is said to be *secure* if there exists efficient (poly-time) algorithms ("simulators") S_A, S_B and S_C such that for any three inputs a, b, c

$$\left\{S_A(f_A(a, b, c), 1^k)\right\}_{k\in N} \stackrel{c}{\equiv} \left\{\mathsf{VIEW}_A^{(A,B,C)}(a, b, c, 1^k)\right\}_{n\in N}$$

$$\left\{S_B(f_B(a, b, c), 1^k)\right\}_{k\in N} \stackrel{c}{\equiv} \left\{\mathsf{VIEW}_B^{(A,B,C)}(a, b, c, 1^k)\right\}_{k\in N}$$

$$\left\{S_C(f_B(a, b, c), 1^k)\right\}_{k\in N} \stackrel{c}{\equiv} \left\{\mathsf{VIEW}_C^{(A,B,C)}(a, b, c, 1^k)\right\}_{k\in N}$$

where $\stackrel{c}{\equiv}$ denotes computational indistinguishability.

The implication of the existence of an efficient simulator is that the party whose view can be efficiently simulated does not learn anything from the protocol's execution beyond what could be inferred from the output of the functionality. In other words, the protocol does not reveal anything beyond the output, which is what one would have expected if the functionality would have been realized via a trusted-third party (or a black-box for that matter). This follows from the fact that any party attempting to attack the protocol and learn something from the interaction might as well have run the simulator (while getting the protocol's output directly from a trusted third party) without having to engage in any interaction with the other parties.

Since communication is done via SSL, then the messages sent between any two parties (and in particular those that do participate in the protocol) can be assumed to be authenticated and secret with respect to other (and in particular non-participating) parties. In particular, simulation of messages encrypted under SSL and not intended to a certain party becomes straightforward, and consists of merely picking encryptions of

random messages and outputting them as a simulated transcript. This also applies in cases when one of the parties does not actively participate in the protocol execution.

Note that the only protocol in which three parties actually participate is the *New Appliance Instance* protocol. Even in that case, the three party communication is somewhat degenerate, since there is no direct communication between U and P (they communicate through A_i). This means that these parties' views only consist of messages sent to them by A_i (however, A_i's view does consists of messages sent both by U and P).

Since the analysis takes place in the semi-honest model, the simulation of the parties' views in all protocols turns out to be fairly straightforward, and consists of picking the appropriate random strings.

5.2 Assumptions

The assumptions underlying the security of the HKM protocols are:

1. **SSL-generated channels are private and authenticated.** This is a standard assumption and it underlies the security of a significant part of secure internet communication. It should be noted that this assumption will hold only provided that a trusted Public-Key infrastructure is in place (and hence we are implicitly also assuming the existence of the latter).
2. **ElGamal is a secure encryption scheme (equivalent to assuming that DDH holds).** This is a standard assumption and it underlies many standardized schemes.
3. **Parties U, A_i and P are all semi-honest.** That is, they are assumed to follow the prescribed behavior of the protocol (most importantly, whenever required they sample random strings uniformly at random). Note that this does not mean that parties will not attempt to learn information from their view of the transcript, and this is what makes this security guarantee non-trivial to achieve. We additionally assume that no two parties out of U, A_i and P collude to attack the protocol.
4. **SHA-256 implements a random oracle.** This assumption is not mathematically well-defined, but is nevertheless standard in cryptographic protocol design. The SHA-256 function is used to derive actual keys from the specific keys S_j. The assumption that it behaves like a random oracle will enable us to postulate that there is no practical harm in letting the PVKM learn certain relations between specific keys (as such relations will be practically "destroyed" with the application of SHA-256).

As mentioned in Section 2, we propose to enhance the protocol in order to make the protocol resilient to malicious adversaries. This will enable to relax Assumption (3). It is also possible to relax assumption number (4) by using t-wise independent "blinding" for the specific keys (see Section 4.3). This will make the protocol less vulnerable to related-key attacks.

5.3 Protocol Analysis

We now turn to prove the security of the various HKM protocols. We start with the *New Appliance Instance* protocol, which is designed to securely realize the (randomized) functionality $f(K, \lambda, \lambda) = (\lambda, \gamma_i, K \cdot \gamma_i)$ in the semi-honest model, where $\gamma_i \leftarrow G$. Note that, in this functionality, no matter what is the value K that U feeds as input, the outputs of A_i and P are random and individually (though not jointly) independent of K.

Proposition 5.1. *Suppose that assumptions 1-4 from Section 5.2 all hold. Then, the* New Appliance Instance *protocol (P, A_i, U) securely realizes the three-party (randomized) functionality*

$$f(K, \lambda, \lambda) = (\lambda, \gamma_i, K \cdot \gamma_i)$$

in the semi-honest model, where $\gamma_i \leftarrow G$.

Proof. It follows directly from the protocol's description that if the three parties behave as prescribed on inputs (K, λ, λ), their respective outputs indeed equal $(\lambda, \gamma_i, K \cdot \gamma_i)$ for a random string $\gamma_i \leftarrow G$.

We now simulate U's view. The simulator $S_U(\lambda, 1^k)$ samples $sk_i \leftarrow \mathbb{Z}_q$ and $r_i \leftarrow \mathbb{Z}_q$, computes $h_i = g^{sk_i}$, and outputs (r_i, h_i) as its view. It can be seen that S_U's output is identically distributed to U's view in an interaction with A_i (as both consist of (r_i, h_i) that are sampled according to the same distribution).

We next simulate P's view. The simulator $S_P(\delta_i, 1^k)$ samples $sk_i \leftarrow \mathbb{Z}_q$ and $t_i \leftarrow \mathbb{Z}_q$, computes $h_i = g^{sk_i}$, sets $pk_i = (G, g, h_i)$, and outputs $(sk_i, \mathrm{Enc}_{pk_i}(\delta_i; t_i))$ as its view. Notice that h_i is distributed aa prescribed, and that t_i is uniformly distributed (and in particular distributed identically to $r_i + s_i$ in a normal execution of the protocol). It thus follows that the output of S_P is identically distributed to P's view in an honest execution.

We conclude by simulating A_i's view. The simulator $S_{A_i}(\lambda, 1^k)$ samples $sk_i \leftarrow \mathbb{Z}_q$, $s_i, r_i \leftarrow \mathbb{Z}_q$ and $\gamma_i \leftarrow G$, computes $h_i = g^{sk_i}$, sets $pk_i = (G, g, h_i)$ and outputs $(s_i, \gamma_i, h_i, \mathrm{Enc}_{pk_i}(1; r_i))$ as its view (where $1 \in G$). Since sk_i is chosen at random, then the distribution of h_i is as prescribed. Since r_i is chosen at random, and since ElGamal is semantically secure, it follows that for any $K \in G$, the distribution of $(s_i, \gamma_i, h_i, \mathrm{Enc}_{pk_i}(1; r_i))$ (representing S_{A_i}'s output) is computationally indistinguishable from the distribution $(s_i, \gamma_i, h_i, \mathrm{Enc}_{pk_i}(K; r_i))$ (representing A_i's view). □

We now turn to argue about the security of the *New Storage Object* protocol. Since this protocol only involves P and A_i, there is no need to refer to U in the analysis.

Proposition 5.2. *Suppose that assumptions 1-4 from Section 5.2 all hold. Then, the* New Storage Object *protocol (P, A_i) securely realizes the two-party (randomized) functionality*

$$f(\gamma_i, \delta_i) = (S_j, (\delta_i/\gamma_i) \cdot S_j)$$

in the semi-honest model, where $S_i \leftarrow G$.

Proof. It follows directly from the protocol's description that if the two parties behave as prescribed on inputs (γ_i, δ_i), their respective outputs equal $(S_j, (\delta_i/\gamma_i) \cdot S_j)$ for $S_j \leftarrow G$ (and in particular if $\delta_i = K \cdot \gamma_i$ then P's local output equals $K \cdot S_j$).

Simulation of P's view merely consists of sampling $\alpha_j \leftarrow G$ and outputting α_j along with the message 'Request to generate j^{th} specific key'. To simulate A_i's view, simply pick a string $\beta_{ij} \leftarrow G$ and output it. Since in a real protocol execution, the message β_{ij} that A_i receives is random independently of both parties' inputs, the distribution output by the simulator is identical to the distribution of A_i's view. □

Proposition 5.3. *Suppose that assumptions 1-4 from Section 5.2 all hold. Then, the* Set Protected Item *protocol (P, A_i) securely realizes the two-party (randomized) functionality*

$$f(\gamma_i, \delta_i) = (S_j, (\delta_i/\gamma_i) \cdot S_j)$$

in the semi-honest model, where $S_i \leftarrow G$.

Proof. As before, it follows directly from the protocol's description that if the two parties behave as prescribed on inputs (γ_i, δ_i), their respective outputs equal $(S_j, (\delta_i/\gamma_i) \cdot S_j)$ for $S_j \leftarrow G$ (and in particular if $\delta_i = K \cdot \gamma_i$ then P's local output equals $K \cdot S_j$).

Simulation of P's view consists of sampling $\beta_{ij} \leftarrow G$ and outputting it. To simulate A_i's view, pick $S_j \leftarrow G$ and output it. Since in a real execution, the message β_{ij} that P receives is random independently of both parties' input, the distribution output by the simulator is identical to the distribution of P's view. □

References

[1] P. Eronen and H. Krawczyk. RFC 5869 – HMAC-based Extract-and-Expand Key Derivation Function (HKDF). In `http://tools.ietf.org/html/rfc5869`, 2010.