Optimizing Registration Based Encryption

Kelong Cong[1], Karim Eldefrawy[2], and Nigel P. Smart[1,3]

[1] imec-COSIC, KU Leuven, Leuven, Belgium.
[2] SRI International, Menlo Park, California, USA.
[3] Dept. Computer Science, University of Bristol, Bristol, UK.
kelong.cong@esat.kuleuven.be
karim.eldefrawy@sri.com
nigel.smart@kuleuven.be

Abstract. The recent work of Garg et al. from TCC'18 introduced the notion of registration based encryption (RBE). The principal motivation behind RBE is to remove the key escrow problem of identity based encryption (IBE), where the IBE authority is trusted to generate private keys for all the users in the system. Although RBE has excellent asymptotic properties, it is currently impractical. In our estimate, ciphertext size would be about 11 terabytes in an RBE deployment supporting 2 billion users. Motivated by this observation, our work attempts to reduce the concrete communication and computation cost of the current state-of-the-art construction. Our contribution is two-fold. First, we replace Merkle trees with crit-bit trees, a form of PATRICIA trie, without relaxing any of the original RBE efficiency requirements introduced by Garg et al. This change reduces the ciphertext size by 15% and the computation cost of decryption by 30%. Second, we observe that increasing RBE's public parameters by a few hundred kilobytes could reduce the ciphertext size by an additional 50%. Overall, our work decreases the ciphertext size by 57.5%.

1 Introduction

Identity based encryption (IBE), introduced by Shamir [Sha84], allows Alice to encrypt a message to Bob as long as she knows Bob's identity, such as his email address or pseudonym. This notion significantly simplifies the key-management issue of public key encryption [DH76, RSA78, GM82] (PKE) since it removes the need of a public key infrastructure (PKI). Starting with the first concrete instantiation by Boneh and Franklin [BF01], a long line of research has developed many IBE instantiations from a variety of assumptions. Generalizations of IBE such as attribute based encryption (ABE) [SW05] and functional encryption (FE) [BSW11] have also been recently studied.

Despite the success of the research community in developing practical IBE, IBE has not replaced public key encryption due to the key escrow problem. In an IBE scheme, there exists a key-generation authority that generates decryption keys for every user enrolled in the system. Users must fully trust such an authority to behave honestly, since it has the ability to decrypt every (private) message that it captures. In an age where end-to-end encryption is widely deployed[4], requiring a central authority that can eavesdrop on private communication is considered a major downgrade in security.

An obvious mitigation to this key escrow problem is to homogeneously distribute the power of the key generation authority, which was already suggested by Boneh and Franklin [BF01]. The work of Kate and Goldberg [KG10], for example, presented a solution based on distributed key generation. Another approach is to distribute the authority heterogeneously. In the work of Chow [Cho09],

[4] WhatsApp uses end-to-end encryption and has 2 billion users [Fac20].

authentication and key-issuance are performed by two different authorities, identity certifying authority and key generation authority. This approach ensures that the key generation authority, which has the master private key, does not know the identities of users. Without the identities, the key generation authority cannot decrypt the messages as long as the the two authorities do not collude. While these ideas mitigate the escrow problem, they do not solve it completely since the authority (or a collective authority) still has the ability to eavesdrop on users.

In another direction, Al-Riyami and Paterson [AP03] put forward the notion of Certificateless Public Key Cryptography; there is no escrow problem and there is no need for certificates in such schemes. Thus, this notion can be seen as a hybrid between PKE+PKI and IBE. Unfortunately, it does not have the convenient features of IBE since users cannot encrypt messages using only the identities of the receivers (assuming some known system-wide public parameter).

1.1 Registration Based Encryption: Prior Work

Motivated by these problems, Garg et al. [GHMR18] initiated the study of Registration-Based Encryption (RBE) where the authority does not hold any secret and is fully transparent. At a high level, in the definition of Registration-Based Encryption (RBE) [GHMR18], every user registers their public key and identity with a key curator (KC). Compared to a typical IBE scheme, the KC does not generate decryption keys nor does it hold any secret information; it simply acts as an accumulator. Although the KC may sound like a PKI, it does not answer user queries for public keys. Instead, it publishes a relatively short public parameter that *every* user can use to perform encryption. Similar to IBE, the encryptor only needs to know the identity of the receiver and the short public parameter to generate a ciphertext. The decryptor needs some "supporting information", that does not need to be kept secret from the KC, and its *own* private key to decrypt. The public parameters in some sense "encode" identities and public keys of all users, in a highly compact form.

To make RBE more attractive, the definition of RBE [GHMR18] formulates the following efficiency requirements:

1. The public parameters must be short, i.e., $\mathsf{poly}(\lambda, \log n)$, where n is the number of registered users and λ is the security parameter.
2. The registration process and the generation of supporting information must be efficient, i.e., they must run in time $\mathsf{poly}(\lambda, \log n)$ per user registration.
3. The number of times that a decryptor must request supporting information must be low, i.e., $\mathsf{poly}(\lambda, \log n)$ over the lifetime of the system.

Below we give an overview of the RBE literature without describing them in detail. However, see Section 2 for a gentle introduction to the blueprint which all constructions follow. The detailed explanation is deferred to when we describe our contribution, since constructions share a similar blueprint.

The authors of [GHMR18] described a construction based on indistinguishability obfuscation (iO) [BGI$^+$01, GGH$^+$13] and somewhere statistically-binding hash functions (SSBH) [HW15] which achieves all the efficiency requirements. They also proposed a weakly efficient construction based on standard assumptions but the registration process must run in time $\mathsf{poly}(\lambda, n)$.

Followup work [GHM$^+$19] solved the issue above and introduced the first RBE scheme that satisfies all the efficiency requirements from standard assumptions. Their 'efficient' RBE construction is achieved via a a two-step approach, where they used the construction of [GHMR18] to bootstrap the fully efficient construction. Further, the authors introduce anonymous RBE which requires that the ciphertext generated on a uniformly random message looks uniformly random (irrespective of the recipient).

An outstanding security issue is that the KC could maliciously register duplicate identities with different public keys where it knows the corresponding secret key. A malicious user could do the same if the KC does not check for uniqueness. This behavior essentially gives the the attacker a trapdoor, allowing him to read messages that are encrypted for an honestly registered user. The same attack also applies to PKI systems. Motivated by the above, the third work on RBE [GV20] studied the verifiability aspect and described an efficient construction where the user who has identity id can ask the KC to prove that id is unique. Further, the authors introduced the "snapshot trick" that removed the bootstrapping step of previous constructions [GHM+19].

1.2 Our Contributions

As mentioned above, existing RBE constructions already achieve very appealing asymptotic complexity, i.e., short public parameters, and efficient generation and requesting of (updated) supporting information, that scale with $\mathsf{poly}(\lambda, \log n)$, where n is the number of registered users and λ is the security parameter. Unfortunately, the requirement to garble public key operations, which is a key building block in all existing RBE schemes, makes such schemes impractical.

Concretely, suppose this operation is implemented using elliptic curve cryptography, for example based on secp192k1 [Cer10]. One (garbled) curve multiplication in this case requires 19.2 billion non-XOR gates[5] and 366 gigabytes of communication [JLE17]. Worse, this operation is performed $O(\log n)$ times, where n is the number of users. For example, using the most efficient construction [GV20], Alice would need to send approximately 11 terabytes to Bob if there are 2 billion users and λ is 256 bits. Undoubtedly, for RBE to be of practical use, we need to focus on reducing the concrete computation and communication cost. To this end, we make the following contributions that aim to improve the concrete efficiency.

1. We introduce an authenticated version of crit-bit trees [Ber] (which might be of independent interest), a form of authenticated PATRICIA trie [Mor68]. We use authenticated crit-bit trees instead of Merkle trees in our RBE construction. This modification reduces the number of input bits of the circuits that we need to garble, which directly decreases the number of public key encryption circuits. We estimate a 15% reduction in computation and communication (ciphertext size) by the encryptor and a 30% reduction in computation by the decryptor. Our construction preserves the verifiability property introduced in [GV20].
2. Furthermore, we suggest a modification to the RBE public parameter which reduces computation and communication of the encryptor by a half, in addition to the improvement above. However, this modification requires us to relax the compactness requirement in typical RBE schemes from $\mathsf{poly}(\lambda, \log n)$ to $O(\lambda, \sqrt{n})$, where n is the number of users registered in the system and λ is the security parameter. For many applications, we argue that this is a reasonable assumption since the total number of users would reach a saturation point, eventually. For example, WhatsApp uses end-to-end encryption and has 2 billion users [Fac20]; with an n of 2 billion our construction would only add 187 kilobytes to the public parameters.

[5] Free-XOR [KS08] is an optimization for garbled circuits which allows the garbler to create the garbled truth table "for free", without symmetric key operations.

With these two main optimizations we estimate a 57.5% reduction (on average) in the communication cost, i.e., ciphertext size. Although our contribution does not make RBE practical, we believe it is a significant step in the right direction. A promising future work could study public key operations that are garbled-circuit friendly. The communication cost could be significantly reduced if such a primitives exist.

Our work follows the original RBE security definition [GHMR18] which does not include a decryption oracle. In other words, we do not handle active attacks, this limitation is not unique to our scheme, existing RBE constructions in the literature exhibit the same limitation. Defining and designing an RBE scheme that is secure under chosen ciphertext attacks is still currently an open question and left for future work.

2 Registration Based Encryption: A Tutorial

Before giving the formal definitions, we describe the key idea behind all RBE construction using a series of strawman constructions so that the readers who are unfamiliar with RBE can build an intuition of how it works and why the key building block, hash garbling, is needed. We begin our discussion by considering a fix set of users, of size n, and only focus on the encryption and decryption functionality. Then we describe the more dynamic setting where new users are allowed to register.

2.1 Encryption and Decryption

Consider three parties, the encryptor Alice, the decryptor Bob and the key curator (KC). Alice wants to send an encrypted message to Bob using only Bob's identity, e.g., his email address, and a short public parameter pp provided by the KC. Bob should be able to decrypt the message using his secret key (which he generated by himself) and some short, non-secret "supporting information" u, provided by the KC. What follows is a series of strawman constructions that we will refine one at a time. Eventually, we will arrive at a construction that is very close to what is described in the literature.

Strawman 1 (RBE from iO): Let the KC store a Merkle tree where the intermediate nodes have the form $(\mathsf{id}^*\|\alpha\|\beta)$, where id^* is the largest identity[6] of the left sub-tree, α is the digest of the left node and β is the digest of the right node. The digests are computed using some hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$, where λ is the security parameter. For example, $\alpha_1 = \mathsf{H}(\mathsf{id}_2^*\|\alpha_2\|\beta_2)$ in Figure 1. The leaf nodes store the user identities and their corresponding public keys, i.e., they have the form $(0^\lambda\|\mathsf{id}\|\mathsf{pk})$. For brevity, we assume the tree is perfectly balanced, i.e., the number of leaves is a power of two. We denote the depth of the tree with d and the Merkle root by rt. The public parameter is $\mathsf{pp} \leftarrow (\mathsf{rt}, d)$ and we let the u data of Bob be the authenticating path from the root to the leaf that contains Bob's identity.

The reason for storing the identity of the left sub-tree is so that the KC can search for an identity in $O(\log n)$ time, using the binary search algorithm, when a decryptor asks for his path u. Consequently, the identities stored in the leaves must be sorted.

Another ingredient we need for encryption and decryption is a circuit P. This circuit takes Bob's u as input, checks whether u is a valid path that begins with rt and ends with a leaf node

[6] We assume the identities can be ordered.

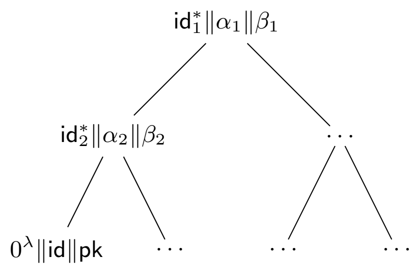

$\mathsf{id}_1^*\|\alpha_1\|\beta_1$
$\mathsf{id}_2^*\|\alpha_2\|\beta_2$
$0^\lambda\|\mathsf{id}\|\mathsf{pk}$

Fig. 1. The Merkle tree structure used for RBE. The leaves are sorted by id. Consider a node, id^* is the identity of the largest identity in its left sub-tree.

containing Bob's identity, and finally outputs $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$ where pk is Bob's public key taken from the leaf node. For a path to be valid we require that, for every node $(\mathsf{id}^*\|\alpha\|\beta)$, the hash of the child node is β if Bob's identity is greater than id^* otherwise α. We write $P(u; m, \mathsf{pp})$ for the circuit, where m and pp is hardwired into the circuit and u is the undetermined input.

If Alice wants to send an encrypted message m to Bob using only pp and Bob's identity, she creates an obfuscated version of the circuit $\tilde{P} \leftarrow \mathsf{Obf}(P(?; m, \mathsf{pp}))$ and then sends it to Bob. Upon receiving $\tilde{P}$, Bob simply evaluates it to obtain ct and attempts to decrypt it using his secret key sk. Note that anyone who has the path u, e.g., the KC, can evaluate $\tilde{P}$, but it is not possible to recover the underlying message m since they do not have Bob's secret key.

To achieve ciphertext indistinguishability, we need to argue that two obfuscated programs with different hardwired m are indistinguishable. However, indistinguishability obfuscation (iO) only guarantees that the two obfuscated programs are indistinguishable if they have the same functionality which is not the case here. Fortunately, [GHMR18, Theorem 4.3] states that one can achieve indistinguishability for this particular type of program P. The proof relies on the semantic security of the PKE scheme used in the program.

Strawman 2 (Replacing iO using GC): Strawman 1 already has most of ingredients of a typical RBE scheme and it is essentially the idea of the very first RBE construction from [GHMR18]. While it works, it needs to assume that iO exists, which is not a standard assumption.

In the second strawman, we replace the iO idea with a garbling scheme [BHR12, GMW87]. The garbling scheme has two algorithms. The first algorithm, Garble, takes a circuit (e.g., P) as input, and then outputs the garbled circuit (GC) $\tilde{P}$ and all the input labels $\mathbf{k}$, two labels for every bit in the input of P. The evaluation algorithm, Eval, evaluates the GC using the labels that correspond to the evaluator's input x (denoted using $\mathbf{k}_x$). Everything goes correctly when $\mathsf{Eval}(\tilde{P}, \mathbf{k}_x) = P(x)$. A detailed definition is given in Section 3.2.

For this construction to be secure, we need to make a modification to the circuit P. Before the values m and pp are hardwired. But this is not secure since two garbled circuits with different different topologies (different message m) are not indistinguishable. Thus we need to modify $P(u; m, \mathsf{pp})$ to become $P(u, m; \mathsf{pp})$, where m is also undetermined.

Using a garbling scheme, Alice creates a GC and input labels $(\tilde{P}, \mathbf{k}) \leftarrow \mathsf{Garble}(P)$ and then sends $\tilde{P}$ to Bob. But we run into a problem when Bob attempts to evaluate $\tilde{P}$ on the undetermined input (u, m), since he does not have the input labels. The wire labels corresponding to m can be sent along with the ciphertext, but for Bob to obtain the wire labels corresponding to his input u it seems that we require interaction. In the next two strawman constructions we show how to resolve this issue.

Strawman 2.5 (Breaking up the large circuit): Removing interaction is not trivial. Thus, we need to take an intermediate step where we break up one large circuit P into many smaller ones, one for every level of the tree. Below we give a description of a circuit that does *not* work, but it illustrates the idea of what we want to achieve.

For every level $j \in [d]$ in the Merkle tree, the corresponding circuit P_j will have the following logic.

1. Take a Merkle tree node $u_j = (\mathsf{id}_j^* \| \alpha_j \| \beta_j)$ as input.
2. Check whether $\mathsf{H}(u_j) = \mathsf{y}$ where y is some hardwired value, if the check fails then abort.
3. If $\mathsf{id}^* = \alpha_j$, return $\mathsf{Enc}(\beta_j, m)$. Recall that α_j stores the identity and β_j stores the corresponding public key in the leaf node.
4. Else, return the labels that correspond to the preimage of α_j if $\mathsf{id} > \mathsf{id}_j^*$, otherwise return the labels that correspond to β_j.

If Alice sends these circuits to Bob, and if Bob also has the input labels $\mathbf{k}_{u_1}$ then he can evaluate every circuit. That is because the output of every circuit is the input label for the next circuit. The final output is $\mathsf{Enc}(\beta_j, m)$, which Bob can decrypt using his secret key.

The reader may have already notice that in Item 2 above, Alice does not have the digest y to create these circuits. Further, in Item 4, Alice also does not have the preimage of α_j or β_j to generate the labels. Nevertheless, this impossible strawman construction illustrates the idea of "chaining" GCs such that the output of one is used as the input of the next. This idea is crucial for understanding the final strawman construction.

Strawman 3 (Putting everything together using hash garbling): In the final strawman, we realise the idea of chaining GCs and remove the need for interaction using an important primitive called hash garbling, first introduced in [GHMR18].

One can think of hash garbling as an extension to a garbling scheme with an $\mathsf{HG.Input}$ algorithm and a modified evaluation algorithm $\mathsf{HG.Eval}$. $\mathsf{HG.Input}$ outputs *encrypted* labels, denoted by $\tilde{\mathbf{k}}$, given some input y. These encrypted labels are crafted in a way that the actual labels that represent y are only revealed if the preimage, under H, of y is known. Next, $\mathsf{HG.Eval}$ is a modified Eval, it still takes the garbled circuit $\tilde{P}$, but also takes the encrypted labels $\tilde{\mathbf{k}}$ and the preimage x. As we noted earlier, it is only possible to evaluate $\tilde{P}$ on input y if $\mathsf{H}(x) = \mathsf{y}$. We formally define hash garbling in Section 3.3.

Now we are ready to put everything together in the final strawman. We modify the circuit P_j as follows.

1. Take a Merkle tree node $u_j = (\mathsf{id}_j^* \| \alpha_j \| \beta_j)$ as input.
2. If $\mathsf{id} = \alpha_j$, return $\mathsf{Enc}(\beta_j, m)$.
3. Else, return $\mathsf{HG.Input}(\alpha_j)$ if $\mathsf{id} > \mathsf{id}_j^*$ otherwise return $\mathsf{HG.Input}(\beta_j)$.

Alice creates a garbled circuits $\tilde{P}_j$ as before, but she also runs $\tilde{\mathbf{k}}_1 \leftarrow \mathsf{HG.Input}(\mathsf{rt})$, and then sends $(\{\tilde{P}_j\}_{j\in[d]}, \tilde{\mathbf{k}}_1)$ to Bob. Upon receiving the message, Bob begins to evaluate the first circuit with $\mathsf{HG.Eval}(\tilde{P}_1, \tilde{\mathbf{k}}_1, u_1)$. Suppose Bob's identity is in the left sub-tree, the output of $\mathsf{HG.Eval}$ becomes

$\tilde{\mathbf{k}}_2 \leftarrow \mathsf{HG.Input}(\alpha_1)$. Bob continues the evaluation by running $\mathsf{HG.Eval}(\tilde{P}_2, \tilde{\mathbf{k}}_2, u_2)$, and so on. Eventually, Bob obtains $\mathsf{Enc}(\mathsf{pk}, m)$ which he can decrypt using his secret key. An illustration of this process is in Figure 2.

$$\begin{tikzcd}
\tilde{\mathbf{k}}_2 \leftarrow \mathsf{HG.Eval}(\tilde{P}_1, \tilde{\mathbf{k}}_1, u_1) \arrow[d, bend left] \\
\tilde{\mathbf{k}}_3 \leftarrow \mathsf{HG.Eval}(\tilde{P}_2, \tilde{\mathbf{k}}_2, u_2) \arrow[d, bend left] \\
\tilde{\mathbf{k}}_4 \leftarrow \mathsf{HG.Eval}(\tilde{P}_3, \tilde{\mathbf{k}}_3, u_3) \arrow[d, bend left] \\
\mathsf{Enc}(\mathsf{pk}, m) \leftarrow \ldots\ldots
\end{tikzcd}$$

Fig. 2. Illustration of RBE decryption.

The construction is complete. To understand why it removes interaction and correctly checks the Merkle path, we make the following two observations.

1. Interaction is no longer needed in this construction. Alice does not need to stay online after sending a series of garbled circuits and an encrypted input label.
2. Due to the properties of hash garbling, the evaluator must input the correct preimage and encrypted labels to $\mathsf{HG.Eval}$ at every step, otherwise the circuits cannot output the ciphertext at the end. In other words, the evaluator cannot generate the encrypted labels by himself which forces him to use the correct preimage x in $\mathsf{HG.Eval}$, i.e. the path provided to the evaluator by the KC.

2.2 Adding Registration

Let us now consider the issue of registration. Performing registration the naive way, i.e., adding a new leaf node to the Merkle tree whenever a new user registers, would lead to $O(n)$ updates for the supporting information u, the path that contains his leaf.

A simple idea, first introduced in [GHMR18], is to keep multiple Merkle trees. Whenever a new user registers, a new Merkle tree with a single leaf is created. Then, trees that have the same number of leaves are merged to form a new tree. Observe that the sizes of the trees are unique powers of two. The KC needs to publish $O(\log n)$ Merkle roots, so the public parameters are kept small. The encryptor Alice needs to run the encryption procedure for every tree since she does not know where Bob's leaf is. The number of updates that Bob needs to do for u is reduced to $O(\log n)$ due to the following reason. His identity must be in a tree with 2^i leaves, for some integer i. He needs to update u whenever 2^i users are registered in the system after him. When that happens, his identity will be in a tree with 2^{i+1} leaves and the process repeats.

Unfortunately, the registration idea above does not guarantee $\mathsf{poly}(\lambda, \log n)$ computational complexity. Concretely, whenever two Merkle trees are merged, their leaves need to be re-sorted. As such, it is not possible to merge to trees in time $\mathsf{poly}(\lambda, \log n)$ while keeping the leaves sorted.

Subsequent works [GHM$^+$19, GV20] resolved this issue. We use the same idea in our construction thus we defer the details to Section 4.2.

3 Preliminaries

We first present definitions used throughout this paper.

3.1 Public Key Encryption

A public key encryption (PKE) scheme consists of the following PPT algorithms.

- $\mathsf{KGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$. The key generation algorithm takes the security parameter as input, and then outputs a public key and a private key.
- $\mathsf{Enc}(\mathsf{pk}, m) \to \mathsf{ct}$. The encryption algorithm takes a public key pk and a message m as input, and then outputs a ciphertext ct. Sometimes we write $\mathsf{Enc}(\mathsf{pk}, m; r)$ to explicitly specify the randomness $r \in \{0,1\}^\lambda$.
- $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \to m$. The decryption algorithm takes a secret key sk and a ciphertext ct as input, and then outputs a message m.

Definition 1. *(Correctness of PKE) A PKE scheme is correct if for all λ, $m \in \mathcal{M}$ and $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$, it holds that*

$$\Pr[\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = m].$$

Definition 2. *(IND-CPA security of PKE) The PKE scheme is IND-CPA secure if there exists a negligible function $\mathsf{negl}(\lambda)$ such that any PPT adversary $\mathcal{A}$ wins the following game with probability $\frac{1}{2} + \mathsf{negl}(\lambda)$.*

- *The challenger $\mathcal{C}$ generates $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(\lambda)$ and sends pk to $\mathcal{A}$.*
- *$\mathcal{A}$ picks two messages m_0, m_1 and sends them to $\mathcal{C}$.*
- *$\mathcal{C}$ samples $b \leftarrow\!\!\$\ \{0,1\}$ and sends $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, m_b)$ to $\mathcal{A}$.*
- *$\mathcal{A}$ outputs b' and wins if $b = b'$.*

3.2 Garbled Circuits

To build a hash garbling scheme, we also need garbled circuits (GC). We review Yao's GC next using the notation adapted from [BLSV18]. A garbling scheme consist of the following two algorithm.

- $\mathsf{Garble}(1^\lambda, 1^n, 1^m, C, \mathsf{state}) \to (\tilde{C}, \{\mathsf{k}_{i,b}\}_{i \in [n], b \in \{0,1\}})$ is a deterministic algorithm that generates the input labels $\{\mathsf{k}_{i,b}\}_{i \in [n], b \in \{0,1\}}$ as well as all the intermediate labels using state as the seed, and then creates the garbled circuit $\tilde{C}$, which has an input length of n bits and an output length of m bits.
- $\mathsf{Eval}(1^\lambda, \tilde{C}, \{\mathsf{k}_{i,x_i}\}_{i \in [n]})$ evaluates the garbled circuit using the given input labels k_{i,x_i}. Each label corresponds to a bit of the desired input x.

Definition 3. *(Correctness of garbling) For all circuits C, inputs x and secret state $\mathsf{state} \in \{0,1\}^\lambda$, correctness holds when $\mathsf{Eval}(1^\lambda, \tilde{C}, \mathbf{k}) = C(x)$, where $(\tilde{C}, \{\mathsf{k}_{i,b}\}_{i \in [n], b \in \{0,1\}}) \leftarrow \mathsf{Garble}(1^\lambda, 1^n, 1^m, C, \mathsf{state})$ and $\mathbf{k} \leftarrow \{\mathsf{k}_{i,x_i}\}_{i \in [n]}$.*

Definition 4. *(Security of garbling) For any circuit* $C : \{0,1\}^n \to \{0,1\}^m$*, input* $x \in \{0,1\}^n$ *and secret state* $\mathsf{state} \in \{0,1\}^\lambda$*, there exists a simulator* Sim *such that the following distributions are computationally indistinguishable:*

$$\{(\tilde{C}, \tilde{\mathbf{k}}) : (\tilde{C}, \{\tilde{\mathsf{k}}_{i,b}\}_{i\in[n], b\in\{0,1\}}) \leftarrow \mathsf{Garble}(1^\lambda, 1^n, 1^m, C, \mathsf{state}), \tilde{\mathbf{k}} \leftarrow \{\tilde{\mathsf{k}}_{i,x_i}\}_{i\in[n]}\}$$
$$\overset{\mathsf{c}}{\approx}\{(\tilde{C}, \tilde{\mathbf{k}}) : (\tilde{C}, \tilde{\mathbf{k}}) \leftarrow \mathsf{Sim}(1^\lambda, 1^{|C|}, 1^n, C(x))\}.$$

3.3 Hash Garbling

The main ingredient in RBE is hash garbling; it is first introduced in [GHMR18] and then used in a similar manner in subsequent works on RBE [GHM+19, GV20]. In this section, we review its definition from the literature. A hash garbling scheme is defined by the following five algorithms $\mathsf{HG.Gen}$, $\mathsf{HG.Hash}$, $\mathsf{HG.Garble}$, $\mathsf{HG.Input}$, $\mathsf{HG.Eval}$:

- $\mathsf{HG.Gen}(1^\lambda, 1^n) \to \mathsf{hk}$. This algorithm takes a security parameter λ and an input length parameter n, and outputs a hash key hk.
- $\mathsf{HG.Hash}(\mathsf{hk}, x) \to \mathsf{y}$. This is a deterministic algorithm that takes a hash key hk and a preimage $x \in \{0,1\}^n$ as input, and outputs a digest $\mathsf{y} \in \{0,1\}^\lambda$.
- $\mathsf{HG.Garble}(\mathsf{hk}, C, \mathsf{state}) \to \tilde{C}$. This algorithm takes a hash key hk, a circuit C and a secret state $\mathsf{state} \in \{0,1\}^\lambda$ as input, and outputs a garbled circuit $\tilde{C}$ (without labels).
- $\mathsf{HG.Input}(\mathsf{hk}, \mathsf{y}, \mathsf{state}) \to \tilde{\mathbf{k}}$. This algorithm takes a hash key hk, a value $\mathsf{y} \in \{0,1\}^\lambda$ and a secret state $\mathsf{state} \in \{0,1\}^\lambda$ as input, and outputs encrypted labels $\tilde{\mathbf{k}}$.
- $\mathsf{HG.Eval}(\tilde{C}, \tilde{\mathbf{k}}, x) \to z$. This algorithm takes a garbled circuit $\tilde{C}$, encrypted labels $\tilde{\mathbf{k}}$ and a value $x \in \{0,1\}^n$, and outputs a value z.

Definition 5. *(Correctness of Hash Garbling) For all* λ*,* n*, hash key* $\mathsf{hk} \leftarrow \mathsf{HG.Gen}(1^\lambda, 1^n)$*, circuit* C*, input* $x \in \{0,1\}^n$*,* $\mathsf{state} \in \{0,1\}^\lambda$*, garbled circuit* $\tilde{C} \leftarrow \mathsf{HG.Garble}(\mathsf{hk}, C, \mathsf{state})$ *and* $\tilde{\mathbf{k}} \leftarrow \mathsf{HG.Input}(\mathsf{hk}, \mathsf{HG.Hash}(\mathsf{hk}, x), \mathsf{state})$*, we require that*

$$\mathsf{HG.Eval}(\tilde{C}, \tilde{\mathbf{k}}, x) = C(x).$$

Definition 6. *(Security of Hash Garbling) There exists a PPT simulator* Sim *such that for all* λ, n *and PPT adversary* $\mathcal{A}$ *we have*

$$(\mathsf{hk}, x, \tilde{\mathbf{k}}, \tilde{C}) \overset{\mathsf{c}}{\approx} (\mathsf{hk}, x, \mathsf{Sim}(\mathsf{hk}, x, C(x), 1^{|C|})),$$

where hash key $\mathsf{hk} \leftarrow \mathsf{HG.Gen}(1^\lambda, 1^n), (C, x) \leftarrow \mathcal{A}(\mathsf{hk}), \mathsf{state} \leftarrow \{0,1\}^\lambda$*, garbled circuit* $\tilde{C} \leftarrow \mathsf{HG.Garble}(\mathsf{hk}, C, \mathsf{state})$ *and* $\tilde{\mathbf{k}} \leftarrow \mathsf{HG.Input}(\mathsf{hk}, \mathsf{HE.Hash}(\mathsf{hk}, x), \mathsf{state})$*.*

Readers who are interested in more details about hash garbling should refer to [DG17, DGHM18, BLSV18] for constructions based on CDH, factoring, and LWE. In Appendix A we give a construction based on the Decision Diffie–Hellman (DDH) problem in a finite Abelian group, which may help the reader understand the specific details.

3.4 Hash Encryption

Hash Encryption is the key building block needed to construct Hash Garbling schemes. We use the definition from [DGHM18, Definition 9].

- $\mathsf{HE.Gen}(1^\lambda, 1^n) \rightarrow \mathsf{hk}$. This is the key generation algorithm that takes a security parameter and and a output length parameter, and then outputs a hash key hk.
- $\mathsf{HE.Hash}(\mathsf{hk}, x) \rightarrow \mathsf{y}$. This algorithm takes a hash key hk and some input $x \in \{0,1\}^n$ and outputs a digest y.
- $\mathsf{HE.Enc}(\mathsf{hk}, (\mathsf{y}, i, b), m) \rightarrow \mathsf{ct}$. This is the encryption algorithm that takes a hash key hk, a value y, an integer $i \in [n]$, a bit b and a message m, and then outputs a ciphertext ct.
- $\mathsf{HE.Dec}(\mathsf{hk}, x, \mathsf{ct}) \rightarrow \{m, \perp\}$. This is the decryption algorithm that takes a hash key hk, a value x and a ciphertext ct, and then outputs a message m if the decryption is successful, otherwise it outputs $\perp$.

Definition 7. *(Correctness of Hash Encryption) For all $x \in \{0,1\}^n$ and $i \in [n]$, correctness holds when*

$$\Pr[\mathsf{HE.Dec}(\mathsf{hk}, x, \mathsf{HE.Enc}(\mathsf{hk}, (\mathsf{HE.Hash}(\mathsf{hk}, x), i, x_i), m)) = m] \geq 1 - \mathsf{negl}(\lambda),$$

where $\mathsf{hk} \leftarrow \mathsf{HE.Gen}(1^\lambda, 1^n)$ and x_i denotes the ith bit of x.

Definition 8. *(Security of Hash Encryption) The security is defined using the game $\mathsf{IND}^{\mathsf{HE}}$ shown below. The hash encryption scheme is secure when, for any PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$,*

$$\left|\frac{1}{2} - \Pr[\mathsf{IND}^{\mathsf{HE}}(1^\lambda, \mathcal{A}) = 1]\right| \leq \mathsf{negl}(\lambda).$$

```
$\mathsf{IND}^{\mathsf{HE}}(\mathcal{A})$
1 : $(x, \mathsf{state}_1) \leftarrow \mathcal{A}_1(1^\lambda)$
2 : $\mathsf{hk} \leftarrow \mathsf{HE.Gen}(1^\lambda, 1^n)$
3 : $(i \in [n], m_0, m_1, \mathsf{state}_2) \leftarrow \mathcal{A}_2(\mathsf{state}_1, \mathsf{hk})$
4 : $b \leftarrow\$ \{0,1\}$
5 : $\mathsf{ct} \leftarrow \mathsf{HE.Enc}(\mathsf{hk}, (\mathsf{HE.Hash}(\mathsf{hk}, x), i, 1 - x_i), m_b)$
6 : $b' \leftarrow \mathcal{A}_3(\mathsf{state}_2, \mathsf{ct})$
7 : return if $b = b'$ then $1$ else $0$
```

In contrast to witness encryption, where the ciphertext can only be decrypted if the preimage is known, hash encryption has the extra property that the ith bit of x must be b. Batch encryption schemes, described in [BLSV18], can be used to construct hash garbling schemes as well. This fact is shown in [GHM+19].

3.5 Registration Based Encryption

We recall the original definition of registration based encryption (RBE) [GHMR18]. An RBE scheme involves two types of parties. The first is the key curator (KC) that maintains a public parameter pp and some auxillary information aux. The second is the user which can register with the KC and then communicate privately with other users using only the identity of the recipient and pp.

RBE consists of six PPT algorithms: $\mathsf{RBE.Setup}$, KGen, $\mathsf{RBE.Reg}^{[\mathsf{aux}]}$, $\mathsf{RBE.Enc}$, $\mathsf{RBE.Upd}^{\mathsf{aux}}$ and $\mathsf{RBE.Dec}$. The aux superscript means that the algorithm associated with it has read access to the auxiliary information aux. Having a bracket around aux means that it is mutable by the associated algorithm.

- $\mathsf{RBE.Setup}(1^\lambda) \to \mathsf{crs}$. This is the common reference string (CRS) generation algorithm which outputs a CRS crs based on the security parameter λ.
- $\mathsf{KGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$. This is the key generation algorithm of the underlying PKE scheme.
- $\mathsf{RBE.Reg}^{[\mathsf{aux}]}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, \mathsf{pk}) \to \mathsf{pp}'$. The registration algorithm takes a CRS crs, a public parameter pp, an identity id and a its corresponding public key pk as input. It outputs a new public parameter pp'. This algorithm has read and write oracle access to the auxillary information aux.
- $\mathsf{RBE.Enc}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, m) \to \mathsf{ct}$. The encryption algorithm takes as input the CRS crs, public parameters pp, an identity id of the recipient, and a message m, and then outputs a ciphertext ct that encrypts m.
- $\mathsf{RBE.Upd}^{\mathsf{aux}}(\mathsf{pp}, \mathsf{id}) \to u$. The update algorithm takes as input the current public parameter pp stored at the KC and an identity id, and then outputs some information $u \in \{0,1\}^*$ that would help the user who has the identity id with decryption. This algorithm has read-only oracle access to aux.
- $\mathsf{RBE.Dec}(\mathsf{sk}, u, \mathsf{ct}) \to \{m, \perp, \mathtt{GetUpd}\}$. The decryption algorithm takes as input a secret key sk, decryption information u and a ciphertext ct, and then it outputs either a message m, an error $\perp$ or $\mathtt{GetUpd}$ which indicates that u is out of date.

$\mathsf{RBE.Reg}^{[\mathsf{aux}]}$ and $\mathsf{RBE.Upd}^{\mathsf{aux}}$ are *deterministic* algorithm executed by the KC. This property implies that the KC is fully auditable. The other algorithms are randomized.

Next we recall the definition of completeness, compactness, and efficiency from the literature. We use $\mathsf{Comp}_{\mathcal{A}}^{\mathsf{RBE}}$ to define the definitions. It is a game where the adversary $\mathcal{A}$ can register non-target identities and a target identity, and then make encryption and decryption requests.

Definition 9. *(Completeness, compactness, and efficiency of RBE) For any stateful, interactive computationally bounded adversary $\mathcal{A}$ that has a $\mathsf{poly}(\lambda)$ round complexity, consider the following game $\mathsf{Comp}_{\mathcal{A}}^{\mathsf{RBE}}$ between $\mathcal{A}$ and a challenger $\mathcal{C}$.*

1. ***Initialization.*** *The challenger $\mathcal{C}$ initializes parameters as*

$$(\mathsf{pp}, \mathsf{aux}, S_{\mathsf{id}}, \mathsf{id}^*, t) = (\epsilon, \epsilon, \epsilon, \emptyset, \perp, 0),$$

   samples $\mathsf{crs} \leftarrow \mathsf{RBE.Setup}(1^\lambda)$ and sends crs to $\mathcal{A}$. S_{id} is the set of registered identities, id^* is the target identity and t acts as a counter for the number of decryption attempts.
2. ***Query phase.*** *$\mathcal{A}$ makes polynomially many queries of the following form, where each query is considered as a single round of interaction between $\mathcal{C}$ and $\mathcal{A}$.*
   (a) ***Registering a non-target identity.*** *On a query of the form $(\mathsf{regnew}, \mathsf{id}, \mathsf{pk})$, $\mathcal{C}$ checks that $\mathsf{id} \notin S_{\mathsf{id}}$. It aborts if the check fails. Otherwise, $\mathcal{C}$ registers $(\mathsf{id}, \mathsf{pk})$ by running the registration algorithm $\mathsf{RBE.Reg}^{[\mathsf{aux}]}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, \mathsf{pk})$. It adds id to the set as S_{id}. After every query, $\mathcal{C}$ updates the parameters $\mathsf{pp}, \mathsf{aux}, S_{\mathsf{id}}$.*
   (b) ***Registering target identity.*** *On a query of the form $(\mathsf{regtgt}, \mathsf{id})$, $\mathcal{C}$ first checks if $\mathsf{id}^* = \perp$. Again, it aborts if the check fails. Otherwise, $\mathcal{C}$ sets $\mathsf{id}^* \leftarrow \mathsf{id}$, samples a challenge key pair $(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{KGen}(1^\lambda)$, updates the public parameter (and aux) using $\mathsf{pp} \leftarrow \mathsf{RBE.Reg}^{[\mathsf{aux}]}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}^*, \mathsf{pk}^*)$ and inserts id^* into S_{id}. We remark that the challenger stores the secret key sk^* in addition to updating all other parameters. Also, note that the adversary here is restricted to make such a query at most once, since the challenger would abort otherwise.*
   (c) ***Target identity encryption.*** *On a query of the form (enctgt, m), $\mathcal{C}$ checks if $\mathsf{id}^* \neq \perp$. If the check fails, abort. Otherwise, it sets $t \leftarrow t + 1$, $\tilde{m}_t \leftarrow m$, and computes ciphertext $\mathsf{ct}_t \leftarrow \mathsf{RBE.Enc}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}^*, \tilde{m}_t)$. It stores[7] the tuple $(t, \tilde{m}_t, \mathsf{ct}_t)$ and then sends the ct_t to $\mathcal{A}$.*

[7] If $\mathcal{C}$ stores a tuple, it means appending the tuple to $\mathcal{C}$'s local state so that it can be accessed later.

(d) ***Target identity decryptions.*** *On a query of the form* (dectgt, j)*,* $\mathcal{C}$ *checks if* $\mathsf{id}^* \neq \perp$ *and* $j \in [t]$*. If the check fails, abort. Otherwise,* $\mathcal{C}$ *computes* $y_j \leftarrow \mathsf{RBE.Dec}(\mathsf{sk}^*, u, \mathsf{ct}_j)$*. If* $y_j = \mathtt{GetUpd}$*, then it computes* $u \leftarrow \mathsf{RBE.Upd}^{\mathsf{aux}}(\mathsf{p}, \mathsf{id}^*)$ *and then recomputes* $y_j \leftarrow \mathsf{Dec}(\mathsf{sk}^*, u, \mathsf{ct}_j)$*. Finally,* $\mathcal{C}$ *stores the tuple* (j, y_j)*.*

3. ***Output Phase.*** *We say that* $\mathcal{A}$ *wins the game if there is some* $j \in [t]$ *for which* $\tilde{m}_j \neq y_j$*.*

Let $n \leftarrow |S_{\mathsf{id}}|$ *denote the number of identities registered until a specific round in the game above. We require the following properties to hold for any* $\mathcal{A}$ *at any moment during the game* $\mathsf{Comp}_{\mathcal{A}}^{\mathsf{RBE}}$*.*

- ***Completeness.*** $\Pr[\mathcal{A}$ *wins* $\mathsf{Comp}_{\mathcal{A}}^{\mathsf{RBE}}(\lambda)] \leq \mathsf{negl}(\lambda)$*.*
- ***Compactness.*** $|\mathsf{pp}|, |u|$ *are both* $\leq \mathsf{poly}(\lambda, \log n)$*.*
- ***Efficiency of registration and update.*** *The time complexity of each invocation of* $\mathsf{RBE.Reg}^{[\mathsf{aux}]}$ *and* $\mathsf{RBE.Upd}^{\mathsf{aux}}$ *is at most* $\mathsf{poly}(\lambda, \log n)$*.*
- ***Efficiency of the number of updates.*** *The total number of invocations of* $\mathsf{RBE.Upd}^{\mathsf{aux}}$ *for identity* id^* *during the decryption phase is at most* $\mathsf{poly}(\lambda, \log n)$ *for every* n*.*

Definition 10. *(Security of RBE) For any interactive PPT adversary* $\mathcal{A}$*, consider the game* $\mathsf{Sec}_{\mathcal{A}}^{\mathsf{RBE}}(\lambda)$ *below. The definition is similar to the IND-CPA public key encryption definition except that* $\mathcal{A}$ *can register one target and polynomially many non-target identities.*

1. ***Initialization.*** *The challenger* $\mathcal{C}$ *initializes parameters as*

$$(\mathsf{pp}, \mathsf{aux}, u, S_{\mathsf{id}}, \mathsf{id}^*) = (\epsilon, \epsilon, \epsilon, \emptyset, \perp),$$

samples $\mathsf{crs} \leftarrow \mathsf{RBE.Setup}(1^\lambda)$ *and sends* crs *to* $\mathcal{A}$*.*

2. ***Query Phase.*** $\mathcal{A}$ *makes polynomially many queries of the following form.*
   (a) ***Registering non-target identity.*** *On a query of the form* $(\mathsf{regnew}, \mathsf{id}, \mathsf{pk})$*,* $\mathcal{C}$ *checks that* $\mathsf{id} \notin S_{\mathsf{id}}$*. It aborts if the check fails. Otherwise,* $\mathcal{C}$ *registers* $(\mathsf{id}, \mathsf{pk})$ *by running the registration algorithm* $\mathsf{RBE.Reg}^{[\mathsf{aux}]}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, \mathsf{pk})$*. It adds* id *to the set as* S_{id}*. Note that* $\mathsf{pp}, \mathsf{aux}$ *and* S_{id} *is updated after every query.*
   (b) ***Registering target identity.*** *On a query of the form* $(\mathsf{regtgt}, \mathsf{id})$*,* $\mathcal{C}$ *first checks if* $\mathsf{id}^* = \perp$*. Again, it aborts if the check fails. Otherwise,* $\mathcal{C}$ *sets* $\mathsf{id}^* \leftarrow \mathsf{id}$*, samples a challenge key pair* $(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{KGen}(1^\lambda)$*, updates the* pp *and* aux *using* $\mathsf{pp} \leftarrow \mathsf{RBE.Reg}^{[\mathsf{aux}]}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}^*, \mathsf{pk}^*)$ *and inserts* id^* *into* S_{id}*. Finally,* $\mathcal{C}$ *sends* pk^* *to* $\mathcal{A}$*.*
3. ***Challenge Phase.*** *On a query of the form* $(\mathsf{chal}, \mathsf{id}, m_0, m_1)$*, the challenger checks whether* $\mathsf{id} \notin S_{\mathsf{id}} \setminus \{\mathsf{id}^*\}$*. If the check fails, abort. Otherwise,* $\mathcal{C}$ *samples* $b \in \{0, 1\}$ *and computes the challenge ciphertext* $\mathsf{ct} \leftarrow \mathsf{RBE.Enc}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, m_b)$*.*
4. ***Output Phase.*** $\mathcal{A}$ *outputs a bit* b' *and wins the game if* $b' = b$*.*

We say that an RBE scheme is message-hiding secure if for every PPT $\mathcal{A}$ *and every* $\lambda \in \mathbb{N}$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\Pr[\mathcal{A} \text{ wins } \mathsf{Sec}_{\mathcal{A}}^{\mathsf{RBE}}(\lambda)] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

3.6 Crit-bit Tree

One of the key building blocks in our optimized RBE construction is crit-bit trees. We describe a crit-bit tree by comparing it to the trie structure. Tries look like binary trees but the path for searching and inserting an item depends on the binary encoding of the item. For example to find the value $3_{10} = 011_2$, the algorithm would take the path "right, right, left", assuming 0 represents "left" and 1 represents "right". This idea implies that there are at least two types of nodes, (1) intermediate nodes only hold pointers to their children and (2) leaf nodes hold the actual values. An example is given in Figure 3.

PATRICIA Trie [Mor68], also known as radix tree, is a type of trie where some of the intermediate nodes are "compressed". The idea is simple: nodes on a path that do not branch are compressed into one internal node. There are many varieties of PATRICIA tries. In this work, we use crit-bit trees [Ber]. The name comes from "critical bit", which is an integer stored in all the internal nodes that indicates the next bit location where two items differ. Typically, this integer increases with depth. The main reason behind this choice is that crit-bit trees have very small node size, using only two pointers and an integer of size at most the log of bit-length of the leaf size.

An example is given in Figure 4. Note that the two internal nodes in the standard trie on Figure 3 are compressed into one internal node in the crit-bit example. Suppose we want to find the value 100_2, the search algorithm first visits the root node and sees a critical bit of 0, and then decides to go left since the 0th bit (the LSB) of 100_2 is 0. Then the algorithm reaches an intermediate node with a critical bit of 2, it would decide to go right since the 2nd bit of 100_2 (the MSB) is 1.

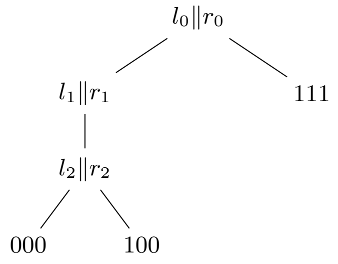


Fig. 3. An example of a trie. Each node has two pointers l_d, r_d which refer to the left or the right child, where d is the depth.

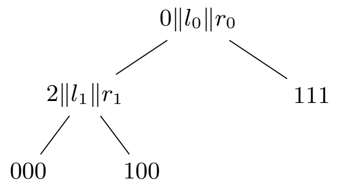


Fig. 4. An example of a crit-bit tree. In addition to the two pointers which it inherited from the trie structure, every node contains a positive integer which represents the "critical bit".

Key properties of PATRICIA tries, which also apply to crit-bit trees, include the following. For random items, the average depth is approximately $\log n + 0.33279$ [Knu98, Page 507]. Szpankowski computed the variance of the depth [Szp90, Equation 2.9] which turned out to be a constant for a fixed branching factor. In the binary case, the variance is 1. We are also interested in the worstcase performance. In the literature, the maximum depth is called the height. The expected value of the height is $\log n + \sqrt{2 \log n} + O(1)$ [KS02, DFHN17]. The final $O(1)$ term is small, typically ranging between 1 and -1 [KS02]. Many of the results above are confirmed experimentally in the literature [NT02].

In Section 4.1, we will describe how a crit-bit tree is augmented to be an authenticated crit-bit tree and give algorithms for searching and inserting in such an authenticated crit-bit tree.

4 Optimizing RBE Using Crit-bit Trees

In this section we describe our optimized RBE construction based on crit-bit trees. Before describing the construction, we define the authenticated crit-bit tree by drawing inspiration from CONIKS [MBB+15] which uses a similar construction but based on tries.

4.1 Authenticated Crit-bit Tree

Assume λ is a power of 2, every node in the tree has $1 + \log \lambda + 2\lambda$ bits and has the format

$$(\tau \in \{0,1\} \parallel \sigma \in \{0,1\}^{\log \lambda} \parallel \alpha \in \{0,1\}^{\lambda} \parallel \beta \in \{0,1\}^{\lambda}),$$

where τ represents the node type and σ represents the critical bit index. For clarity, we let $\mathtt{I} \leftarrow 0$ and $\mathtt{L} \leftarrow 1$. The tree consists of two types of nodes:

1. the intermediate node has the form $(\mathtt{I} \| \sigma \| \alpha \| \beta)$, where α and β correspond to the digest of the left child and the right child, respectively;
2. the leaf node holds the registered user and has the form $(\mathtt{L} \| 0^{\log \lambda} \| \mathsf{id} \| \mathsf{pk})$.

Unlike CONIKS [MBB+15], we do not need an "empty" node because having an empty node implies that there is a path that has no branches, which would be compressed in crit-bit trees.

Authentication is performed in a manner similar to Merkle-tree. Namely, the pointers described in Section 3.6 are replaced by hash pointers. For example, the α value of an internal node is $\mathsf{H}(\mathtt{L} \| 0^{\log \lambda} \| \mathsf{id} \| \mathsf{pk})$ if its left child is a leaf node, where H is a hash function. In our RBE construction (Section 4.2), we use $\mathsf{HG.Hash}$ as the hash function.

Search: The search algorithm follows directly from the crit-bit tree definition. We give a high level description based on [Lan08]. Before giving the search algorithm, we define an algorithm that walks down the tree to find the node that is "closest" to the target identity id. If id exists, then the leaf node containing id is returned. We call this algorithm the "walk algorithm".

1. Let id be the input and we use the $\mathsf{id}[i]$ notation to access the ith bit, $\mathsf{id}[0]$ represents the LSB of id.
2. Let $\mathsf{currNode}$ be the root node, recursively perform the following steps until $\mathsf{currNode}$ is a leaf node.
   (a) Determine the traversal direction, i.e., $\mathsf{dir} \leftarrow \mathsf{id}[\mathsf{currNode}.\sigma]$.
   (b) If $\mathsf{dir} = 0$, set $\mathsf{currNode}$ to the left child, otherwise set it to the right child.
3. Output $\mathsf{currNode}$.

The search algorithm is simply an equality test added to the algorithm above.

1. Run the "walk algorithm" above and obtain a leaf node.
2. Output the leaf node if the leaf node contains id, otherwise ouput $\perp$.

Insertion: The insertion algorithm is a bit more involved because we need make sure the critical bits are always increasing with depth. We give a high level description based on [Lan08] for inserting $(\mathsf{id}, \mathsf{pk})$.

1. Create a leaf node $\mathsf{newLeaf} \leftarrow (\mathtt{L} \| 0^{\log\lambda} \| \mathsf{id} \| \mathsf{pk})$.
2. Find the closest leaf node to id using the "walk algorithm" from above and call it $\mathsf{closestLeaf}$.
3. Starting at the LSB, let σ^* be the critical bit between id and $\mathsf{closestLeaf}.\alpha$.
4. From the root, walk the tree in the same way (using id as the target) and stopping at a node, which we call pNode, if
   (a) it is a leaf node, or
   (b) the critical bit is greater than σ^*.
5. Compute the direction $\mathsf{dir} \leftarrow \mathsf{id}[\sigma^*]$.
6. Create a new internal node $(\mathtt{I} \| \sigma^* \| \alpha \| \beta)$, where $\alpha = \mathsf{H}(\mathsf{newLeaf})$ if $\mathsf{dir} = 0$, otherwise $\alpha = \mathsf{H}(\mathsf{pNode})$. The other digest β is set the same way except it's the mirror image of α. In essence, the new internal node took the position of pNode, and pNode and $\mathsf{newLeaf}$ are its two children.
7. Traverse back up the tree to the root and recompute the digests.

4.2 Optimized RBE Construction with Compact Public Parameters

Tree Structure: Similar to the work of [GV20], our construction uses two data structures, IDTree and CBTree. The first is IDTree, which is a self-balancing binary tree (e.g., Red–black trees) used for internal book-keeping by the KC. Concretely, the nodes have the form (id, t) where id is a user identity and t is a timestamp which always increments by 1. The re-balancing operation is based on the order of id, thus we assume the identities have an ordering.

The second is CBTree, which are crit-bit trees. These trees have the structure describe in Section 4.1. We use ℓ to denote the total number of such trees at any moment in time.

Optimized RBE Construction: Now we are ready to detail our RBE construction. Most of the algorithms follow a similar idea as [GV20] but are adapted to use crit-bit trees. In particular, the registration algorithm is functionally the same as the one in [GV20] but the description is simplified using the critical bit idea.

The KC holds public parameters $\mathsf{pp} = (\mathsf{crs}, \{\mathsf{rt}_i, d_i\}_{i\in[\ell]})$ and auxillary information $\mathsf{aux} = \{\mathsf{IDTree}, \{(\mathsf{CBTree}_i, n_i)\}_i\}$, where rt_i is the digest of the root node of CBTree_i and d_i is the maximum depth of CBTree_i.

– $\mathsf{RBE.Setup}(1^\lambda) \to \mathsf{crs}$. Let $\mathsf{hk} \leftarrow \mathsf{HG.Gen}(1^\lambda, 1^{1+\log\lambda+2\lambda})$. Output hk as crs. Note that the reason for using $1^{1+\log\lambda+2\lambda}$ is because the preimage of the hash function, which is the size of a node, has $1 + \log\lambda + 2\lambda$ bits.
– $\mathsf{KGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$. Generate a public and a secret key pair $(\mathsf{pk}, \mathsf{sk})$ using the public key generation algorithm.
– $\mathsf{RBE.Reg}^{[\mathsf{aux}]}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, \mathsf{pk}) \to \mathsf{pp}'$. The registration is described in the steps below. An example can be found in Figure 5.
  1. Let $(\mathsf{IDTree}, \{(\mathsf{CBTree}_i, n_i)\}_{i\in[\ell]}) \leftarrow \mathsf{aux}$, and let $n = \sum_{i\in[\ell]} n_i$.
  2. Insert $(\mathsf{id}, n + 1)$ to IDTree and call it IDTree'.
  3. Copy the latest tree CBTree_ℓ and call it $\mathsf{NewTree}$, and then insert the leaf $(\mathtt{L} \| 0^{\log\lambda} \| \mathsf{id} \| \mathsf{pk})$ into $\mathsf{NewTree}$.

4. Find the critical bit index σ between n and $n+1$ counting from the MSB.
5. Set $T \leftarrow \{(\mathsf{CBTree}_i, n_i) : i \in [\ell], n_i > 2^\sigma\} \cup \{(\mathsf{NewTree}, 2^\sigma)\}$.
6. Let the new auxillary information be $\mathsf{aux}' \leftarrow \{\mathsf{IDTree}', T\}$.
7. Finally, the KC sets the new public parameter $\mathsf{pp}' \leftarrow (\mathsf{crs}, \{(\mathsf{rt}'_i, d'_i)\}_{i \in [|T|]})$, where rt'_i and d'_i are the new Merkle root and the maximum depth of the trees in T, respectively.

– $\mathsf{RBE.Enc}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, m) \to \mathsf{ct}$. The encryption algorithm uses a program $P_{i,j}$ which we describe first. For clarity, we use Greek alphabets to denote the values that are unknown to the encryptor. The others are constants.

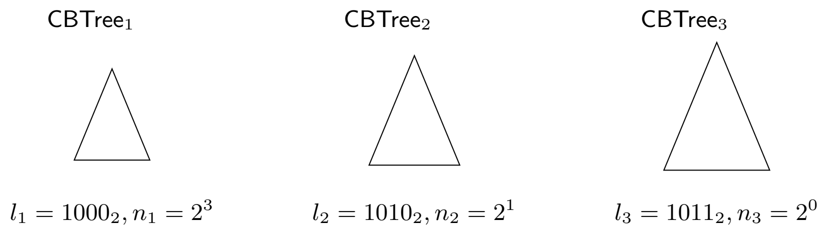
```
$P_{i,j}(\tau \| \sigma \| \alpha \| \beta)$ [Constants: $\mathsf{crs}, \mathsf{state}_{i,j+1}, \mathsf{id}, m, r$]
1 : if $\tau = \mathtt{I}$ then
2 :   if $\mathsf{id}[\sigma] = 0$
3 :     return $\mathsf{HG.Input}(\mathsf{crs}, \alpha, \mathsf{state}_{i,j+1})$
4 :   else
5 :     return $\mathsf{HG.Input}(\mathsf{crs}, \beta, \mathsf{state}_{i,j+1})$
6 :   endif
7 : elseif $\tau = \mathsf{L} \wedge \mathsf{id} = \alpha$ then
8 :   return $\mathsf{Enc}(\beta, m; r)$
9 : else
10 :   return $\perp$
11 : endif
```

Using the program above, the encryption algorithm works as follows.
1. Sample a random value $r \in \{0,1\}^\lambda$.
2. Parse pp as $(\mathsf{hk}, \{(\mathsf{rt}_1, d_1), \ldots, (\mathsf{rt}_\ell, d_\ell)\})$.
3. For each tree index $i \in [\ell]$ and each depth $j \in \{1, \ldots, d_i\}$ of the ith tree, sample $\mathsf{state}_{i,j} \leftarrow\$ \{0,1\}^\lambda$, and then execute
$$\tilde{P}_{i,j} \leftarrow \mathsf{HG.Garble}(\mathsf{hk}, P_{i,j}, \mathsf{state}_{i,j}).$$
4. For every root rt_i, compute $\tilde{\mathbf{k}}_{i,1} \leftarrow \mathsf{HG.Input}(\mathsf{hk}, \mathsf{rt}_i, \mathsf{state}_{1,j})$.
5. Output the ciphertext $\mathsf{ct} = (\mathsf{pp}, \{\tilde{P}_{i,j}\}_{i,j}, \{\tilde{\mathbf{k}}_{i,1}\}_i)$.

– $\mathsf{RBE.Upd}^{\mathsf{aux}}(\mathsf{pp}, \mathsf{id}) \to u$. Let $\mathsf{aux} = (\mathsf{IDTree}, \{(\mathsf{CBTree}_i, n_i)\}_{i \in [\ell]})$ and $\mathsf{pp} = \{(\mathsf{rt}_i, d_i)\}_{i \in [\ell]}$, the update algorithm works as follows.
1. The algorithm performs a binary search in IDTree to find the timestamp t associated with id. If the timestamp does not exist, the algorithm aborts.
2. Otherwise, the algorithm computes an index $i \in [\ell]$ such that
$$\sum_{j \in [i-1]} n_j < t \le \sum_{j \in [i]} n_j.$$
The index i represents the smallest tree index that contains id.
3. Finally, traverse CBTree_i to find the identity and output the traversed path as $u \leftarrow (u_1, \ldots, u_{d_{\mathsf{id}}})$. We use d_{id} to indicate the depth of the path containing id which may be less than the maximum depth of CBTree_i.

– $\mathsf{RBE.Dec}(\mathsf{sk}, u, \mathsf{ct}) \to \{m, \perp, \mathtt{GetUpd}\}$.
1. Let $(u_1, \ldots, u_{d_{\mathsf{id}}}) \leftarrow u$, where u_1 is a root node and $u_{d_{\mathsf{id}}}$ should be $(\mathsf{L} \| 0^{\log_2 \lambda} \| \mathsf{id} \| \mathsf{pk})$ and d_{id} is the depth of the leaf node $u_{d_{\mathsf{id}}}$. If u does not exist, output $\mathtt{GetUpd}$.

2. Let $(\mathsf{pp}, \{\tilde{P}_{i,j}\}_{i,j}, \{\tilde{\mathbf{k}}_{i,1}\}_i) \leftarrow \mathsf{ct}$.
3. Let i be the tree index such that $\mathsf{rt}_i = \mathsf{HG.Hash}(\mathsf{hk}, u_1)$. If no such i exists then output `GetUpd`.
4. For $j \in [d_{\mathsf{id}}]$:
   - Compute $\tilde{\mathbf{k}}_{i,j+1} \leftarrow \mathsf{HG.Eval}(\tilde{P}_{i,j}, \tilde{\mathbf{k}}_{i,j}, u_j)$.
   - If $\tilde{\mathbf{k}}_{i,j+1} = \perp$ then output $\perp$.
5. The final $\tilde{\mathbf{k}}_{i,d_{\mathsf{id}}+1}$ is the ciphertext, so the algorithm decrypts it using the secret key sk, i.e., $\mathsf{Dec}(\mathsf{sk}, \tilde{\mathbf{k}}_{i,d_{\mathsf{id}}+1}) \rightarrow m$, and finally output m.



CBTree_1
CBTree_2
CBTree_3
$l_1 = 1000_2, n_1 = 2^3$
$l_2 = 1010_2, n_2 = 2^1$
$l_3 = 1011_2, n_3 = 2^0$

Fig. 5. There are three crit-bit trees in this example, each tree contains l_i identities. Further, every tree has n_i users that must use the path in the corresponding CBTree_i to decrypt. The trees can be considered as snapshots where the last one is the latest snapshot that contains all the users, i.e., $n = l_3$. If a new user registers, there will be 1100_2 users. The critical bit between 1011_2 and 1100_2 is 1st bit from the MSB, which suggests that CBTree_1 will be kept but the two others will be replaced by $\mathsf{CBTree}_{\mathsf{new}}$ that has $l_{\mathsf{new}} = 1100_2$ and $n_{\mathsf{new}} = 2^2$, according to the registration algorithm.

4.3 Completeness, Efficiency and Compactness

Using the correctness property of PKE (Definition 1) and hash garbling (Definition 5), the completeness of the RBE scheme, from Definition 9, follows by simply following the construction. We can also prove our scheme is efficient and compact according to Definition 9.

- **Compactness.** For the public parameter, there can be at most $\log n$ roots since there is at most $\log n$ trees in aux, thus the public parameter is compact. The path u is also compact since the number of nodes from a root to any leaf is $O(\log n)$.
- **Efficiency of registration and update.** Our registration algorithm first inserts an item into the self-balancing IDTree, which takes $O(\log n)$ time. Then we make a copy of CBTree_ℓ to produce $\mathsf{NewTree}$ and then insert a new leaf node. Insertion takes time $O(\log n)$ for a crit-bit tree, but implementing the copy operation natively will take $O(n)$ time. Fortunately, we can use techniques such as copy-on-write and only allocate storage for the $O(\log n)$ nodes in $\mathsf{NewTree}$ that are different from CBTree_ℓ since the insertion algorithm (Section 4.1) only modifies nodes on a single path. Finally, finding the critical bit and then selecting which trees to delete takes $O(\log n)$ time. Thus, the overall time complexity for registration is $O(\log n)$.
  The update algorithm finds the timestamp t of id which takes time $O(\log n)$ since IDTree is balanced. Then, the algorithm computes the tree index that contains id which also takes $O(\log n)$ since there are only $\log n$ indices. Finally, finding the correct leaf and outputting the path to the leaf is $O(\log n)$ as well due to the tree structure. Thus, the overall time complexity for the update algorithm is $O(\log n)$.

– **Efficiency of the number of updates.** An identity registered at time t is associated with CBTree_i if
$$\sum_{j\in[i-1]} n_j < t \le \sum_{j\in[i]} n_j,$$
where $n_i \ge 2n_{i+1}$. A user needs to fetch a new u, using $\mathsf{RBE.Upd}^{\mathsf{aux}}$, whenever the tree CBTree_i containing his identity is removed by the registration algorithm. Suppose CBTree_i exists at a moment in time, the registration algorithm only deletes it after n_i new identities are registered after it. In other words, for a particular identity, the n_i value associated to the earliest snapshot CBTree_i that contains the identity will grow in powers of 2 as new users are registered. Thus we conclude that the number of updates needed by any user is $\log n$.

4.4 Security

In this section, we follow the template of [GHMR18] and present the security proof. To build intuition, we begin by presenting a proof for when only one user has registered. Then we move on to the general case.

Proof for One User: Single-user security is defined below, which is essentially ciphertext indistinguishability.

Theorem 1. *(RBE security for one user) For any identity* id *we have*

$$\mathsf{ct}_0 = (\mathsf{HG.Garble}(\mathsf{hk}, P[\mathsf{id}, 0, r], \mathsf{state}), \mathsf{HG.Input}(\mathsf{hk}, \mathsf{rt}, \mathsf{state})) \overset{c}{\approx} (\mathsf{HG.Garble}(\mathsf{hk}, P[\mathsf{id}, 1, r], \mathsf{state}), \mathsf{HG.Input}(\mathsf{hk}, \mathsf{rt}, \mathsf{state})) = \mathsf{ct}_1,$$

where $\mathsf{hk} \leftarrow \mathsf{RBE.Setup}(1^\lambda)$, $\mathsf{state} \leftarrow \{0,1\}^\lambda$, $\mathsf{rt} \leftarrow \mathsf{HG.Hash}(\mathsf{hk}, (\mathsf{L}\|0^\lambda\|\mathsf{id}\|\mathsf{pk}))$, $r \in \{0,1\}^\lambda$, $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$ $m \in \{0,1\}$ *and the circuit* P *is defined below. This circuit is an equivalent but simplified version of* $P_{i,j}$ *in Section 4.2 that works for only one user. We abuse the notation and use* $P[\mathsf{id}, m, r]$ *to indicate the constants used in the circuit.*

```
P(τ‖σ‖α‖β) [Constants: id, m, r]
1 : if $\tau \neq \mathsf{L} \wedge \alpha \neq \mathsf{id}$
2 : else return $\mathsf{Enc}(\beta, m; r)$
3 : endif
```

Proof. For $m \in \{0,1\}$, let ct_m denote the challenge ciphertext, i.e.,

$$\mathsf{ct}_m \leftarrow (\mathsf{HG.Garble}(\mathsf{hk}, P[\mathsf{id}, m, r], \mathsf{state}), \mathsf{HG.Input}(\mathsf{hk}, \mathsf{rt}, \mathsf{state})).$$

We show that $\mathsf{ct}_0 \overset{c}{\approx} \mathsf{ct}_1$. By simulation security of the hash garbling scheme (Definition 6), for $m \in \{0,1\}$, we have

$$\mathsf{ct}_m \overset{c}{\approx} \mathsf{Sim}(\mathsf{hk}, (\mathsf{L}\|0^\lambda\|\mathsf{id}\|\mathsf{pk}), \mathsf{Enc}(\mathsf{pk}, m; r), 1^{|P|}).$$

By semantic security of the public key encryption scheme, we can write

$$\mathsf{Sim}(\mathsf{hk}, (\mathsf{L}\|0^\lambda\|\mathsf{id}\|\mathsf{pk}), \mathsf{Enc}(\mathsf{pk}, 0; r), 1^{|P|}) \overset{c}{\approx} \mathsf{Sim}(\mathsf{hk}, (\mathsf{L}\|0^\lambda\|\mathsf{id}\|\mathsf{pk}), \mathsf{Enc}(\mathsf{pk}, 1; r), 1^{|P|}),$$

which concludes the proof. □

General Proof: Now we are ready to prove RBE security for the general case. Without loss of generality, we will only consider one crit-bit tree. Recall that for encryption, if we have ℓ roots, we create circuits individually for each root. Suppose at the time of encryption, we have ℓ trees with roots $\mathsf{rt}_1, \ldots, \mathsf{rt}_\ell$. Then, between the two hybrids which correspond to an encryption of zero and an encryption of one, we may consider ℓ intermediate hybrids, where under the ith hybrid we encrypt 0 under the roots $\mathsf{rt}_1, \ldots \mathsf{rt}_i$ and we encrypt 1 under the roots $\mathsf{rt}_{i+1}, \ldots \mathsf{rt}_\ell$. Thus, using the hybrid argument above, it is enough to only consider one crit-bit tree.

When only considering one tree, the proof is a straightforward hybrid argument. Recall that the ciphertext contains d garbled programs, one for every level of the tree. Starting with the correctly computed ciphertext. we define a series of hybrids where the garbled program and the garbled input are replaced by the *simulated* version one by one. From the security of the hash garbling scheme, these hybrids are computationally indistinguishable. In the final hybrid, we can switch the underlying plaintext using Theorem 1.

Theorem 2. *Our crit-bit tree based RBE construction is secure with respect to Definition 10.*

Proof. Since we are only considering one tree, we will ignore the tree index. That is, $P_{i,j}$ becomes P_j, $\mathsf{state}_{i,j}$ becomes state_j and so on.

Consider an identity id, the path leading to it is $(u_1, u_2, \ldots, u_d)$, where u_1 is the root node and $u_d = (\mathsf{L}\|0^\lambda\|\mathsf{id}\|\mathsf{pk})$. For $j > 1$, let $\tilde{\mathbf{k}}_j \leftarrow \mathsf{HG.Input}(\mathsf{hk}, u_j, \mathsf{state}_j)$. Now we are ready to give the hybrids.

- **Hybrid 0 (encryption in real game).** Let the ciphertext be $\mathsf{ct}_0 \leftarrow (\tilde{P}_1, \ldots, \tilde{P}_d, \tilde{\mathbf{k}}_1)$, where every value is sampled from the construction.
- **Hybrid 1.** Let $\mathsf{ct}_1 \leftarrow (\hat{P}_1, \ldots, \tilde{P}_d, \hat{\mathbf{k}}_1)$, where we use a circumflex to denote simulated values, i.e.,
$$(\hat{P}_1, \hat{\mathbf{k}}_1) \leftarrow \mathsf{Sim}(\mathsf{hk}, u_1, \tilde{\mathbf{k}}_2, 1^{|P_1|}).$$
The other values are sampled as the construction. Recall that $\tilde{P}_j$ is generated using P_j and state_j in the construction. But in this hybrid, and the ones below, we simulate $\tilde{P}_j$ without P_j or state_j.
- **Hybrid $i \in [d-1]$.** Let $\mathsf{ct}_i \leftarrow (\hat{P}_1, \ldots, \hat{P}_i, \tilde{P}_{i+1}, \ldots, \tilde{P}_d, \hat{\mathbf{k}}_1)$, where for $j \in [i]$ $(\hat{P}_j, \hat{\mathbf{k}}_j) \leftarrow \mathsf{Sim}(\mathsf{hk}, u_{j+1}, \tilde{\mathbf{k}}_{j+1}, 1^{|P_j|})$.
- **Hybrid d.** Let $\mathsf{ct}_d \leftarrow (\hat{P}_1, \ldots, \hat{P}_d, \hat{\mathbf{k}}_1)$, where all the values are simulated like the hybrid above for $j \in [d-1]$ and
$$(\hat{P}_d, \hat{\mathbf{k}}_d) \leftarrow \mathsf{Sim}(\mathsf{hk}, u_d, \mathsf{Enc}(\mathsf{pk}, m; r), 1^{|P_d|}).$$

From the security of hash garbling (Definition 6), any two adjacent hybrids are indistinguishable. In the final hybrid, we use the same argument as Theorem 1, i.e.,

$$\begin{aligned} &\mathsf{Sim}(\mathsf{hk}, (\mathsf{L}\|0^\lambda\|\mathsf{id}\|\mathsf{pk}), \mathsf{Enc}(\mathsf{pk}, 0; r), 1^{|P|}) \\ \overset{c}{\approx}\ &\mathsf{Sim}(\mathsf{hk}, (\mathsf{L}\|0^\lambda\|\mathsf{id}\|\mathsf{pk}), \mathsf{Enc}(\mathsf{pk}, 1; r), 1^{|P|}), \end{aligned}$$

to claim that the ciphertexts are indistinguishable. Hence, the security of our RBE construction is proved. □

4.5 Performance Improvement Over Prior Work

Our main performance improvement comes from reducing the number of bits stored in the tree node from $1 + 3\lambda$ to $1 + \log \lambda + 2\lambda$. The reason this is important is because $\mathsf{HG.Input}$ needs to perform 2 public key operations in the garbled circuit per bit, which is exceptionally costly.

The tradeoff is that the depth is higher than a balanced Merkle tree which has $\lceil \log n \rceil$ depth. The decryption algorithm $\mathsf{RBE.Dec}$ is only affected by the average depth $\log n + 0.33279$ (discussed in Section 3.6) so it is a small price to pay to benefit from crit-bit trees. Suppose $n = 2^{31}$, which is a reasonable number for popular applications considering WhatsApp has 2 billion users [Fac20], and $\lambda = 256$. our construction makes 31% fewer public key operations in the GC compared to the best prior work [GV20] on average.

The encryption algorithm $\mathsf{RBE.Enc}$, however, is affected by the maximum depth which is $\log n + \sqrt{2 \log n} + O(1)$. It tends to $\log n$ as n tends to infinity. This property implies that our encryption performance becomes better as the number of registered user grows. In practice, n is not infinite. Suppose $n = 2^{31}$ and $\lambda = 256$ again, the encryption algorithm in our construction makes 15% fewer public key operations in the GC on average.

The calculations above is purely based on the number of public key operations that must be performed in the GC. For every circuit, we assume the PKE uses one pubic key operation and $\mathsf{HG.Input}$ uses $2 \cdot \lambda$ public key operations. Nevertheless, the circuit that the encryptor needs to garble also contains other operations such as branching and string comparison. But the cost of these operations are negligible as they only require a few logic gates, relatively speaking. Naively, a branching operation can be implemented with a 2-to-1 multiplexer where the input is λ bits using $3\lambda + 1$ gates compared to billions of gates in the case of one public key operation.

In Section 5 we slightly weaken the compactness requirement of RBE and further reduce the number public key operations in the encryption algorithm.

4.6 Verifiability

The work of [GV20] introduced verifiable RBE. This property allows users to request a pre-registration proof and a post-registration proof. The former is a proof of non-membership. The latter is a proof of unique-membership. While we do not give the full verification algorithm and prove its soundness and completeness, we argue that it is possible to add pre/post-registration proofs, which essentially only depend on the authenticity of aux (crit-bit trees in our case), without changing the underlying construction.

Before presenting our argument, we introduce the notion of adjacent paths. A pair of adjacent paths in a crit-bit tree is two valid paths[8] with leaf nodes containing $\mathsf{id}^{(0)}$ and $\mathsf{id}^{(1)}$ such that there does not exist another leaf node with id such that $\mathsf{id}^{(0)} < \mathsf{id} < \mathsf{id}^{(1)}$. Concretely, the two paths have the following form,

- $u^{(0)} = \{(\mathtt{I}\|\sigma_1^{(0)}\|\alpha_1^{(0)}\|\beta_1^{(0)}), \ldots, (\mathtt{I}\|\sigma_k^{(0)}\|\alpha_k^{(0)}\|\beta_k^{(0)}), \ldots, (\mathtt{L}\|0^{\log \lambda}\|\mathsf{id}^{(0)}\|\mathsf{pk}^{(0)})\}$,
- $u^{(1)} = \{(\mathtt{I}\|\sigma_1^{(1)}\|\alpha_1^{(1)}\|\beta_1^{(1)}), \ldots, (\mathtt{I}\|\sigma_k^{(1)}\|\alpha_k^{(1)}\|\beta_k^{(1)}), \ldots, (\mathtt{L}\|0^{\log \lambda}\|\mathsf{id}^{(1)}\|\mathsf{pk}^{(1)})\}$.

For every $i \in [k]$, $u_i^{(0)} = u_i^{(1)}$. That is, the two paths share the same prefix of length k and σ_k is the critical bit that distinguishes $\mathsf{id}^{(0)}$ and $\mathsf{id}^{(1)}$, e.g., $\mathsf{id}^{(0)}[\sigma_k] = 0$ and $\mathsf{id}^{(1)[\sigma_k]} = 1$. Further, for $b \in \{0, 1\}$ and $i \in [k+1, d^{(b)}]$, we require that $\mathsf{id}^{(b)}[\sigma^{(b)}] = 1 - b$, where $d^{(b)}$ is the length of the two

[8] A path is valid when the adjacent nodes obey the hash-pointer constraint.

paths. Intuitively, the two paths diverge after the kth node. But after this point, the left path ($u^{(0)}$) must always follow the *right* branch and the right path $u^{(1)}$ must always follow the *left* branch.

For the non-membership proof in the pre-registration phase, the KC simply constructs the pair of adjacent paths described above to prove id does not exist. That is, constructing a pair of adjacent paths with $\mathsf{id}^{(0)}$ and $\mathsf{id}^{(1)}$ in the leaves such that $\mathsf{id}^{(0)} < \mathsf{id} < \mathsf{id}^{(1)}$. The KC can perform this step efficiently and the prove size is compact, i.e., $O(\log n)$. Clearly, verification is also efficient and runs in time $O(\log n)$. This idea generalizes to multiple trees by making one such proof for every tree.

For the unique-membership proof in the post-registration phase, the KC constructs two pairs of adjacent paths. Suppose we want to prove the uniqueness of id, then the KC constructs one pair of adjacent paths with leaves $\mathsf{id}^{(0)}$ and id, and another with the leaves id and $\mathsf{id}^{(1)}$. As a result, for every identity that is unique registered, the KC is able to create the two pairs of adjacent paths. Extending this idea to multiple trees is a bit different than the pre-registration phase above. Similar to [GV20], we can view the trees as snapshots. Which means the KC needs to produce non-membership proofs for identities that are not yet in the snapshots and unique-membership proof for identities that are in the snapshot.

5 Further Optimization using Larger Public Parameter

Using even fewer public key operations is possible if we relax the compactness requirement of RBE. Concretely, the original definition requires pp to have size $\mathsf{poly}(\lambda, \log n)$. However, if we relax the requirement to $\mathsf{poly}(\lambda, \sqrt{n})$, it is possible to reduce the number of public key operations needed in the GC by a half for the encryptor. In practice, if there are 2^{31} registered users and the identities of $\sqrt{2^{31}}$ users are published using a cuckoo filter [FAKM14] with a false positive rate of 2^{-40}, then the size of the public parameter is only increased by 187 kilobytes. We argue that this is a reasonable tradeoff to make to alleviate the bottleneck.

Below we describe the intuition before detailing the modification of the registration and the encryption algorithm. Starting from the scheme in Section 4.2, recall that aux stores $(\{\mathsf{CBTree}_i, n_i\}_{i\in[\ell]})$, where n_i is a power of 2 representing the number of identities that needs to use a path in CBTree_i to decrypt. For brevity, assume $n_i = 2 \cdot n_{i+1}$ for all $i \in [\ell]$. Then majority of the users only need the first half of the trees $\{\mathsf{CBTree}_i\}_{i\in[\ell/2]}$ to decrypt and only a minority need the second half. If the encryptor knows whether a user belongs to the first half or the second half, then he only needs to iterate over half of the trees in the encryption algorithm. Thus, if we allow the KC to publish the identities that belong the second half of the trees (there will be $O(\sqrt{n})$ of them), then the number public key operations in the GC would be halved. The same argument applies in the general case where $n_i \geq 2 \cdot n_{i+1}$ since the bit-pattern of n is uniformly distributed at any moment in time.

5.1 Optimized RBE Construction with Larger Public Parameters

First we describe the new format of the public parameter and then highlight the changes in the two algorithms. The public parameter now has the form

$$\{\mathsf{IDTree}, \{(\mathsf{rt}_i, d_i, n_i)\}_{i\in[\ell]}, \mathcal{I} = \{I_i : i \in [\ell], n_i < \sqrt{2^{\lfloor \log_2 n \rfloor}}\}\},$$

where I_i represent the set of identities that need a path in CBTree_i to decrypt. Note that $|I_i| = n_i$ and $\lfloor \log_2 n \rfloor$ gives the number of bits of n. We view $\mathcal{I}$ as a flattened set to simplify notation but it can be implemented using a cuckoo filter as mentioned above.

The modified registration and encryption algorithm are shown below. We copy the algorithm verbatim from Section 4.2 and highlight the differences with the asterisk * symbol. The other algorithms remain unchanged.

– $\mathsf{RBE.Reg}^{[\mathsf{aux}]}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, \mathsf{pk}) \to \mathsf{pp}'$.
  1. Let $(\mathsf{IDTree}, \{(\mathsf{CBTree}_i, n_i)\}_{i\in[\ell]}) \leftarrow \mathsf{aux}$, and let $n = \sum_{i\in[\ell]} n_i$.
  2. Insert $(\mathsf{id}, n+1)$ to IDTree and call it IDTree'.
  3. Copy the latest tree CBTree_ℓ and call it $\mathsf{NewTree}$, and then insert the leaf $(\mathsf{L}\|0^{\log\lambda}\|\mathsf{id}\|\mathsf{pk})$ into $\mathsf{NewTree}$.
  4. Find the critical bit index σ between n and $n+1$ counting from the MSB.
  5. Set $T \leftarrow \{(\mathsf{CBTree}_i, n_i) : i \in [\ell], n_i > 2^\sigma\} \cup \{(\mathsf{NewTree}, 2^\sigma)\}$.
  6. Let the new auxillary information be $\mathsf{aux}' \leftarrow \{\mathsf{IDTree}', T\}$.
  *7. Update the set $\mathcal{I}$ according to its definition and call it $\mathcal{I}'$.
  *8. Finally, the KC sets the new public parameter

$$\mathsf{pp}' \leftarrow (\mathsf{crs}, \{(\mathsf{rt}'_i, d'_i, n'_i)\}_{i\in[|T|]}, \mathcal{I}'),$$

  where rt'_i, d'_i and n'_i are the new Merkle root, the maximum depth and the number of users of the trees in T, respectively.

– $\mathsf{RBE.Enc}(\mathsf{crs}, \mathsf{pp}, \mathsf{id}, m) \to \mathsf{ct}$. The encryption algorithm uses a program $P_{i,j}$ which we describe first. For clarity, we use Greek alphabets to denote the values that are unknown to the encryptor. The others are constants.

```
$P_{i,j}(\tau\|\sigma\|\alpha\|\beta)$ [Constants: $\mathsf{crs}, \mathsf{state}_{i,j+1}, \mathsf{id}, m, r$]
 1 : if $\tau = \mathtt{I}$ then
 2 :   if $\mathsf{id}[\sigma] = 0$
 3 :     return $\mathsf{HG.Input}(\mathsf{crs}, \alpha, \mathsf{state}_{i,j+1})$
 4 :   else
 5 :     return $\mathsf{HG.Input}(\mathsf{crs}, \beta, \mathsf{state}_{i,j+1})$
 6 :   endif
 7 : elseif $\tau = \mathsf{L} \wedge \mathsf{id} = \alpha$ then
 8 :   return $\mathsf{Enc}(\beta, m; r)$
 9 : else
10 :   return $\perp$
11 : endif
```

Using the program above, the encryption algorithm works as follows.
  1. Sample a random value $r \in \{0,1\}^\lambda$.
  *2. Parse pp as $(\mathsf{hk}, \{(\mathsf{rt}_1, d_1, n_1), \ldots, (\mathsf{rt}_\ell, d_\ell, n_\ell)\}, \mathcal{I})$.
  *3. Let $L \leftarrow \{1, \ldots, v\}$ if $\mathsf{id} \notin \mathcal{I}$, otherwise let $L \leftarrow \{v+1, \ldots, \ell\}$, where v is an index to a crit-bit tree such that $n_v < \sqrt{2^{\lfloor \log_2 n \rfloor}}$.
  *4. For each crit-bit tree $i \in L$ and each depth $j \in \{1, \ldots, d_i\}$ of the ith tree, sample $\mathsf{state}_{i,j} \leftarrow\!\!\$\ \{0,1\}^\lambda$, and then execute

$$\tilde{P}_{i,j} \leftarrow \mathsf{HG.Garble}(\mathsf{hk}, P_{i,j}, \mathsf{state}_{i,j}).$$

  5. For every root rt_i, compute $\tilde{\mathbf{k}}_{i,1} \leftarrow \mathsf{HG.Input}(\mathsf{hk}, \mathsf{rt}_i, \mathsf{state}_{1,j})$.
  6. Output the ciphertext $\mathsf{ct} = (\mathsf{pp}, \{\tilde{P}_{i,j}\}_{i,j}, \{\tilde{\mathbf{k}}_{i,1}\}_i)$.

5.2 Correctness, Security and Efficiency

Correctness and security hold since this construction is similar to the one given in Section 4.2 except the public parameter has some additional information to help the encryptor select which trees to use. If $\mathsf{id} \in \mathcal{I}$, then the encryption algorithm uses the "smaller" trees, i.e., $\{\mathsf{CBTree}_i : i \in [\ell], n_i < \sqrt{2^{\lfloor \log_2 n \rfloor}}\}$. Otherwise id must be in the "bigger" trees. The construction guarantees that the encryptor will always use roots of the trees that contain id.

The time complexity of the new registration algorithm does not change since updating the set $\mathcal{I}$ can be performed at the same time as selecting which trees to include in the new auxillary information (Item 5). The update algorithm stays the same so the time complexity does not change as well.

However, our public parameters are not compact anymore since $O(\sqrt{n})$ identities are included in them. But we feel this is a good tradeoff, since the additional data can be stored in a compress format cuckoo filter as mentioned at the start.

Acknowledgments

This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203, by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. FA8750-19-C-0502 (Approved for Public Release, Distribution Unlimited), and by the FWO under an Odysseus project GOH9718N. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ERC, DARPA, the US Government or the FWO. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

AP03. Sattam S. Al-Riyami and Kenneth G. Paterson. Certificateless public key cryptography. In Chi-Sung Laih, editor, *Advances in Cryptology – ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 452–473, Taipei, Taiwan, November 30 – December 4, 2003. Springer, Heidelberg, Germany.

Ber. Daniel J. Bernstein. Crit-bit trees. `https://cr.yp.to/critbit.html`. [Online; accessed 10 February 2021].

BF01. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

BGI$^+$01. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

BHR12. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 784–796, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

BLSV18. Zvika Brakerski, Alex Lombardi, Gil Segev, and Vinod Vaikuntanathan. Anonymous IBE, leakage resilience and circular security from new assumptions. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 535–564, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

BSW11. Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273, Providence, RI, USA, March 28–30, 2011. Springer, Heidelberg, Germany.

Cer10. Certicom Research. SEC 2: Recommended elliptic curve domain parameters. `https://www.secg.org/sec2-v2.pdf`, January 2010. [Online; accessed: 26 February 2021].

Cho09. Sherman S. M. Chow. Removing escrow from identity-based encryption. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 256–276, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

DFHN17. Michael Drmota, Michael Fuchs, Hsien-Kuei Hwang, and Ralph Neininger. External profile of symmetric digital search trees (extended abstract). In *2017 Proceedings of the Meeting on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 124–130, 2017.

DG17. Nico Döttling and Sanjam Garg. Identity-based encryption from the Diffie-Hellman assumption. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 537–569, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

DGHM18. Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, and Daniel Masny. New constructions of identity-based and key-dependent message secure encryption schemes. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018: 21st International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 10769 of *Lecture Notes in Computer Science*, pages 3–31, Rio de Janeiro, Brazil, March 25–29, 2018. Springer, Heidelberg, Germany.

DH76. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

Fac20. Facebook. Two billion users–connecting the world privately. `https://about.fb.com/news/2020/02/two-billion-users/`, February 2020. [Online; accessed: 12 February 2021].

FAKM14. Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

GGH$^+$13. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual Symposium on Foundations of Computer Science*, pages 40–49, Berkeley, CA, USA, October 26–29, 2013. IEEE Computer Society Press.

GHM$^+$19. Sanjam Garg, Mohammad Hajiabadi, Mohammad Mahmoody, Ahmadreza Rahimi, and Sruthi Sekar. Registration-based encryption from standard assumptions. In Dongdai Lin and Kazue Sako, editors, *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 11443 of *Lecture Notes in Computer Science*, pages 63–93, Beijing, China, April 14–17, 2019. Springer, Heidelberg, Germany.

GHMR18. Sanjam Garg, Mohammad Hajiabadi, Mohammad Mahmoody, and Ahmadreza Rahimi. Registration-based encryption: Removing private-key generator from IBE. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018: 16th Theory of Cryptography Conference, Part I*, volume 11239 of *Lecture Notes in Computer Science*, pages 689–718, Panaji, India, November 11–14, 2018. Springer, Heidelberg, Germany.

GM82. Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *14th Annual ACM Symposium on Theory of Computing*, pages 365–377, San Francisco, CA, USA, May 5–7, 1982. ACM Press.

GMW87. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 218–229, New York, NY, USA, 1987. Association for Computing Machinery.

GV20. Rishab Goyal and Satyanarayana Vusirikala. Verifiable registration-based encryption. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 621–651, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

HW15. Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In Tim Roughgarden, editor, *ITCS 2015: 6th Conference on Innovations in Theoretical Computer Science*, pages 163–172, Rehovot, Israel, January 11–13, 2015. Association for Computing Machinery.

JLE17. Bargav Jayaraman, Hannah Li, and David Evans. Decentralized certificate authorities. `https://arxiv.org/pdf/1706.03370.pdf`, 2017.

KG10. Aniket Kate and Ian Goldberg. Distributed private-key generators for identity-based cryptography. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10: 7th International Conference on Security in Communication Networks*, volume 6280 of *Lecture Notes in Computer Science*, pages 436–453, Amalfi, Italy, September 13–15, 2010. Springer, Heidelberg, Germany.

Knu98. Donald E Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley Longman, 2 edition, 1998.

KS02. Charles Knessl and Wojciech Szpankowski. Limit laws for the height in patricia tries. *J. Algorithms*, 44(1):63–97, July 2002.

KS08. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498, Reykjavik, Iceland, July 7–11, 2008. Springer, Heidelberg, Germany.

Lan08. Adam Langley. Crit-bit trees. `https://www.imperialviolet.org/binary/critbit.pdf`, September 2008. [Online; accessed: 10 February 2021].

MBB+15. Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 383–398, Washington, DC, USA, August 12–14, 2015. USENIX Association.

Mor68. Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

NT02. Stefan Nilsson and Matti Tikkanen. An experimental study of compression methods for dynamic tries. *Algorithmica*, 33(1):19–33, 2002.

RSA78. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978.

Sha84. Adi Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53, Santa Barbara, CA, USA, August 19–23, 1984. Springer, Heidelberg, Germany.

SW05. Amit Sahai and Brent R. Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473, Aarhus, Denmark, May 22–26, 2005. Springer, Heidelberg, Germany.

Szp90. Wojciech Szpankowski. Patricia tries again revisited. *J. ACM*, 37(4):691–711, October 1990.

A Hash Garbling Construction Based on DDH

Hash Garbling is itself based on Hash Encryption, so we first address constructing a Hash Encryption scheme based on the DDH assumption.

A.1 Hash Encryption Construction

The Diffie-Hellman based instantiation of hash encryption is given below. It is inspired by the Chameleon encryption construction found in [DG17] except that we do not need the trapdoor. Additionally, it is similar to the Computational Diffie-Hellman based batch encryption construction from [BLSV18] except that we do not need the blindness property.

Let $\mathsf{H}(\cdot)$ be a collision-resistant hash function, $(\mathbb{G}, \cdot)$ be a group of order q where the decisional Diffie-Hellman (DDH) problem is hard and let g be its generator. The hash encryption scheme Π is defined as follows.

– $\mathsf{HE.Gen}(1^\lambda, 1^n) \to \mathsf{hk}$: Sample $2n$ group elements $g_{j,b} \leftarrow\$\, \mathbb{G}$ and label them as follows:

$$\mathsf{hk} \leftarrow ((g_{1,0}, g_{1,1}), (g_{2,0}, g_{2,1}), \ldots, (g_{n,0}, g_{n,1})).$$

Output hk.

- $\mathsf{HE.Hash}(\mathsf{hk}, x \in \{0,1\}^n) \to \mathsf{y}$: Output $\prod_{j\in[n]} g_{j,x_j}$
- $\mathsf{HE.Enc}(\mathsf{hk}, (\mathsf{y}, i, b), m) \to \mathsf{ct}$:
  1. Sample $r \leftarrow\$\, \mathbb{Z}_q$.
  2. For $j \in [n] \setminus \{i\}$ and $b \in \{0,1\}$, let $\widetilde{g}_{j,b} \leftarrow g_{j,b}^r$.
  3. For $b \in \{0,1\}$, let $\widetilde{g}_{i,b} \leftarrow \perp$.
  4. Compute $\widetilde{\mathsf{y}} \leftarrow \mathsf{y}^r$.
  5. Compute $e \leftarrow m \oplus \mathsf{H}(g_{i,b}^r)$.
  6. Output $(e, \widetilde{\mathsf{y}}, \{\widetilde{g}_{j,b}\}_{j\in[n], b\in\{0,1\}})$.
- $\mathsf{HE.Dec}(\mathsf{hk}, x, \mathsf{ct}) \to m$:
  1. Let $c \leftarrow \widetilde{\mathsf{y}} / \prod_{j\in[n]\setminus\{i\}} \widetilde{g}_{j,x_j}$.
  2. Output $\mathsf{H}(c) \oplus e$.

Theorem 3. *The correctness property in Definition 7 holds for the DH based hash encryption construction.*

Proof. Computing the first part of the decryption procedure gives us

$$\frac{\widetilde{\mathsf{y}}}{\prod_{j\in[n]\setminus\{i\}} \widetilde{g}_{j,x_j}} = \frac{\prod_{j\in[n]} g_{j,x_j}^r}{\prod_{j\in[n]\setminus\{i\}} g_{j,x_j}^r} = g_{i,x_i}^r.$$

Then we compute $\mathsf{H}(g_{i,x_i}^r) \oplus e$ which gives us m. □

Theorem 4. *The security property in Definition 8 holds for the DH-based hash encryption construction assuming the DDH problem is hard.*

Proof. Recall that the DDH problem is hard with respect to the group $\mathbb{G}$ if for all PPT adversary $\mathcal{A}$ there exists a negligible function $\mathsf{negl}(\lambda)$ such that

$$|\Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^{xy})]| \le \mathsf{negl}(\lambda).$$

Using the above, we want to show

$$\Pr[\mathsf{IND}_{\Pi}^{\mathsf{HE}}(1^\lambda, \mathcal{A}) = 1] \le \frac{1}{2} + \mathsf{negl}(\lambda).$$

Consider a distinguisher $\mathsf{D}(\mathbb{G}, q, g, h_1, h_2, h_3)$ with the following code.

1. Pick a random bits $x_i \in \{0,1\}$ and $b \in \{0,1\}$.
2. Sample $i \in [n]$.
3. $(x^*, \mathsf{state}_1) \leftarrow \mathcal{A}_1(1^\lambda)$.
4. Let $\mathsf{hk} \leftarrow \{(g^{\alpha_{j,0}}, g^{\alpha_{j,1}})\}_{j\in[n]}$, except one element $g_{i,1-x_i} \leftarrow h_1$.
5. $(i^*, m_0, m_1, \mathsf{state}_2) \leftarrow \mathcal{A}_2(\mathsf{state}_1, \mathsf{hk})$.
6. If $i \neq i^*$ or $x_i \neq x_i^*$, output a random bit.
7. Compute $\mathsf{y} \leftarrow \mathsf{HE.Hash}(\mathsf{hk}, x^*)$ as usual.
8. Prepare the ciphertext
   - $\widetilde{\mathsf{y}} \leftarrow h_2^{\sum \alpha_{i,x_i}}$,
   - $\widetilde{g}_{j,b} \leftarrow h_2^{\alpha_{j,b}}$ for $j \in [n] \setminus \{i\}$ and $b \in \{0,1\}$,
   - and $e \leftarrow m_b \oplus \mathsf{H}(h_3)$.

Let $\mathsf{ct} \leftarrow (e, \widetilde{\mathsf{y}}, \{\widetilde{g}_{j,b}\}_{j\in[n],b\in\{0,1\}})$.

9. Query the adversary $b' \leftarrow \mathcal{A}_3(\mathsf{state}_2, \mathsf{ct})$ and output 1 if $b' = b$ otherwise 0.

We ignore the "unlucky" case in step 6 initially but come back to it later. Consider two cases, the first is when D is given a random triple $(h_1 = g^x, h_2 = g^y, h_3 = g^z)$. The key used for encryption, h_3, is independent of h_1 and h_2, the trapdoor $\alpha_{j,b}$ is also uniformly random, so output ciphertext component e is uniformly random at step 8 if $\mathsf{H}(\cdot)$ is modelled as a random oracle. All the other messages $(\mathsf{hk}, \widetilde{\mathsf{y}}, \{\widetilde{g}_{j,b}\}_{j\in[n]\setminus\{i\}})$ given to $\mathcal{A}$ has the correct distribution as the security game. More formally,

$$\Pr[\mathsf{D}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] = \Pr[\mathsf{IND}^{\mathsf{HE}}_{\Pi'}(1^\lambda, \mathcal{A}) = 1] = \frac{1}{2},$$

where Π' is the protocol where we execute $\mathsf{H}(\cdot)$ on a random input.

The second case is when D is given a DDH triple, namely $h_1 = g^x, h_2 = g^x, h_3 = g^{xy}$. Observe that this case correctly simulates the encryption scheme, where every input to $\mathcal{A}$ has the correct distribution as the security game. For instance, the input to $\mathsf{H}(\cdot)$ is $g^{xy} = h_2^{\alpha_{i,x_{1-i}}} = h_1^y = g^y_{i,1-x_i}$, where y is the random exponent (unknown to D) used to prepare the ciphertext. Formally,

$$\Pr[\mathsf{D}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] = \Pr[\mathsf{IND}^{\mathsf{HE}}_{\Pi}(1^\lambda, \mathcal{A}) = 1].$$

Putting the two cases together with the DDH assumption, we arrive at

$$\begin{aligned}
\mathsf{negl}(\lambda) &\geq |\Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^{xy})]| \\
&= \Big| \Big(\frac{n-1}{n} \cdot \frac{1}{2} + \frac{1}{n} \Pr[\mathsf{IND}^{\mathsf{HE}}_{\Pi'}(1^\lambda, \mathcal{A}) = 1] \Big) \\
&\quad - \Big(\frac{n-1}{n} \cdot \frac{1}{2} + \frac{1}{n} \cdot \Pr[\mathsf{IND}^{\mathsf{HE}}_{\Pi}(1^\lambda, \mathcal{A}) = 1] \Big) \Big| \\
&= \Big| \frac{n-1}{n} \cdot \frac{1}{2} + \frac{1}{n} \cdot \frac{1}{2} - \frac{n-1}{n} \cdot \frac{1}{2} - \frac{1}{n} \cdot \Pr[\mathsf{IND}^{\mathsf{HE}}_{\Pi}(1^\lambda, \mathcal{A}) = 1] \Big| \\
&= \Big| \frac{1}{n} \cdot \Big(\frac{1}{2} - \Pr[\mathsf{IND}^{\mathsf{HE}}_{\Pi}(1^\lambda, \mathcal{A}) = 1] \Big) \Big| \\
&= \Big| \frac{1}{2} - \Pr[\mathsf{IND}^{\mathsf{HE}}_{\Pi}(1^\lambda, \mathcal{A}) = 1] \Big|,
\end{aligned}$$

which concludes the proof. □

A.2 Hash Garbling Construction

Below is a hash garbling construction. Correctness and security follows straight forwardly from the definition of hash encryption (Section 3.4) and the garbling scheme(Section 3.2).

- $\mathsf{HG.Gen}(1^\lambda, 1^n) \to \mathsf{hk}$. Run $\mathsf{hk} \leftarrow \mathsf{HE.Gen}(1^\lambda, 1^n)$ and output hk.
- $\mathsf{HG.Hash}(\mathsf{hk}, x) \to \mathsf{y}$. Run $\mathsf{y} \leftarrow \mathsf{HE.Hash}(\mathsf{hk}, x)$ and output y.
- $\mathsf{HG.Garble}(\mathsf{hk}, C, \mathsf{state}) \to \tilde{C}$. Run

$$(\tilde{C}, \{\mathsf{k}_{i,b}\}_{i\in[n],b\in\{0,1\}}) \leftarrow \mathsf{Garble}(1^\lambda, 1^n, 1^m, C, \mathsf{state}),$$

but ignore the labels and only output $\tilde{C}$.

– $\mathsf{HG.Input}(\mathsf{hk}, \mathsf{y}, \mathsf{state}) \to \tilde{\mathbf{k}}$. Run

$$(\tilde{C}, \{\mathsf{k}_{i,b}\}_{i\in[n],b\in\{0,1\}}) \leftarrow \mathsf{Garble}(1^\lambda, 1^n, 1^m, C, \mathsf{state}),$$

but ignore the garbled circuit $\tilde{C}$. For every $i \in [n]$ and $b \in \{0,1\}$, compute

$$\tilde{\mathsf{k}}_{i,b} \leftarrow \mathsf{HE.Enc}(\mathsf{hk}, (\mathsf{y}, i, b), \mathsf{k}_{i,b}).$$

Finally, output the $2 \cdot n$ encrypted labels $\tilde{\mathbf{k}} \leftarrow \{\tilde{\mathsf{k}}_{i,b}\}_{i\in[n],b\in\{0,1\}}$.

– $\mathsf{HG.Eval}(\tilde{C}, \tilde{\mathbf{k}}, x) \to z$. Parse $\tilde{\mathbf{k}}$ as $\{\tilde{\mathsf{k}}_{i,b}\}_{i\in[n],b\in\{0,1\}}$. Let x_i be the ith bit of x. For $i \in [n]$, compute

$$\mathsf{k}_{i,x_i} \leftarrow \mathsf{HE.Dec}(\mathsf{hk}, (x, i), \tilde{\mathsf{k}}_{i,x_i}).$$

Finally, evaluate the circuit $z \leftarrow \mathsf{Eval}(\tilde{C}, \{\mathsf{k}_{i,x_i}\}_{i\in[n]})$ and output z.

In this scheme, it is possible to use different types of label. Concretely, during garbling, we can generate the labels as group elements and hash them before using those labels to garble the circuit. When running HG.Input, we generate the labels as group elements as before and pass them straight to HE.Enc, there is no need to hash them. As a consequence, HG.Eval needs to use group version when running HE.Dec and the hashed version when running Eval.