Attribute Based Encryption with Sublinear Decryption from LWE

Prabhanjan Ananth ∗ Xiong Fan †

Abstract

Attribute based encryption (ABE) is an advanced encryption system with a built-in mechanism to generate keys associated with functions which in turn provide restricted access to encrypted data. Most of the known candidates of attribute based encryption model the functions as circuits. This results in significant efficiency bottlenecks, especially in the setting where the function associated with the ABE key admits a RAM program whose runtime is sublinear in the length of the attribute. In this work we study the notion of attribute based encryption for random access machines (RAMs), introduced in the work of Goldwasser, Kalai, Popa, Vaikuntanathan and Zeldovich (Crypto 2013). We present a construction of attribute based encryption for RAMs satisfying sublinear decryption complexity assuming learning with errors. This improves upon the work of Goldwasser et al., who achieved this result based on SNARKs and extractable witness encryption. However, unlike Goldwasser et al., the parameters in our system (including attribute keys) grow with the attribute length and the worst case runtime bound. We note that even with this size restriction on the keys, it was unclear how to achieve sublinear decryption complexity under standard assumptions. Moreover, our work presents the first construction to achieve input-specific runtime for ABE under standard assumptions.

En route to constructing this primitive, we introduce the notion of controlled homomorphic recoding (CHR) schemes. We present a generic transformation from this primitive to attribute-based encryption for RAMs and then we show how to instantiate controlled homomorphic recoding schemes based on learning with errors (LWE).

1 Introduction

Attribute based encryption [SW05] is a powerful paradigm that provides a controlled access mechanism to encrypted data. Unlike a traditional encryption scheme, in an attribute based encryption scheme, an authority can generate a constrained key sk_P for a program P such that it can decrypt an encryption of message μ associated with attribute x only if the condition P(x) = 0 is satisfied. The last decade of research in this area [SW05, GPSW06, OSW07, GJPS08, Wat09, LW11, Wat12, GVVW15a, GGH+13, GKP+13b, BGG+14, GGHZ14, Wee14, GVVW15b, BV16] has led to several useful applications including verifiable computation [PRV12] and reusable garbled circuits [GKP+13a]. Special cases of ABE, such as identity based encryption [BF01, Wat05, DG17, BLSV17], and generalizations of ABE, such as FE [BSW11, O’N10, GGH+16], have also been extensively studied.

Current known constructions of ABE offer different flavors of efficiency guarantees and can be based on various cryptographic assumptions. Barring few exceptions, all these constructions [GPSW06,
Wat09, LOS+10, GVW15a, BGG+14, GVW15b] model the random access programs, associated with the constrained keys, as circuits. However, transforming random access programs into circuits is associated with significant efficiency costs. If the execution time of these programs were sub-linear in the input length to begin with (for instance, binary search), modeling them as circuits destroys the sub-linearity property. As a consequence, the decryption complexity could be exponential in the running time of the programs. This is quite unsatisfactory as we often encounter scenarios where sublinear computations have to be performed on massive data sets. Even if the programs do not have sublinear complexity in the input length, another issue with modeling programs as circuits is that the decryption algorithm could be drastically slower than the running time of the original programs, not to mention the additional overhead involved in transforming programs into circuits.

To circumvent these issues, Goldwasser et al. [GKP+13b] introduced the notion of ABE for RAMs\footnote{A RAM program is associated with a memory (initialized with the input to the RAM program) and step circuit. In every step of the RAM computation, the step circuit outputs the next index to be read and additionally, it also writes to a location in the memory. It differs from a Turing machine, in that a RAM program does not have to read all the locations in the memory.} that showed how to generate keys for RAM programs directly without having to go through the expensive RAM-to-circuit conversion. They presented a construction of ABE for RAMs based on extractable witness encryption and SNARKs (succinct non-interactive arguments of knowledge). Recent works [GGHW14, BP14, BSW16] have brought into question the veracity of the assumptions of extractable witness encryption and SNARKs. While the existence of these assumptions have been ruled out only in specific scenarios, they certainly guide us to be more careful about using them for cryptographic applications.

1.1 Our Contributions

The goal of this work is to understand the feasibility of achieving ABE with sublinear decryption complexity based on well studied cryptographic assumptions. Before stating our result, we explain the model of ABE for RAMs below.

As defined in an ABE for circuits scheme, an ABE for RAMs scheme consists of setup, key generation, encryption and decryption algorithms. The encryption algorithm takes as input an attribute database $D$, a message $\mu$ and produces the ciphertext $ct$. The key generation takes as input a RAM program $P$ and produces attribute key $sk_P$ associated with $P$. The decryption algorithm, modeled as a RAM program, takes as input $sk_P$, a ciphertext $ct$ and produces the decrypted message $\mu$ only if $P^D$ (this notation means $P$ has oracle access to the database $D$) outputs 0.

We note that the syntax of Goldwasser et al. is slightly different. In particular, in their scheme the key generation takes as input program $P$, database $D$ while the encryption only takes as input $x$. In the end, the decryption succeeds only if $P^D(x) = 0$ and the requirement is that for sub-linear programs, the decryption time is sub-linear in $|D|$. The key efficiency requirement we place on our definition is that the decryption of $sk_P$ on encryption of $\mu$ should take time $p(\lambda, T)$, where $T$ is an upper bound on the running time of $P$, for a fixed polynomial $p(\cdot)$. In particular, if $T$ is polylogarithmic in the length of the attribute, namely $|D|$, then the decryption complexity is also polylogarithmic in $|D|$. We term this sublinear decryption property. Barring the work of Goldwasser et al. [GKP+13b], none of the ABE constructions achieve sublinear decryption complexity property.

We show the following result:
Theorem 1.1 (Informal). Assuming learning with errors (with sub-exponential modulus\(^2\)), there is a construction of public key attribute-based encryption scheme for random access machines satisfying sub-linear decryption property.

We emphasize that our scheme supports a priori unbounded number of attribute keys in the security game (referred to as collusion resistance in literature). Also, our construction is in the selective security setting. Even for programs that don’t have sublinear runtime, our work beats previous works in terms of decryption complexity. We elaborate more on this when we compare our work to previous works.

To prove the above theorem, we introduce a novel primitive that we call controlled homomorphic recoding schemes. This primitive generalizes the concepts of fully key homomorphic encryption, introduced in the work of \cite{BGG+14}. Using this tool, we build ABE for RAMs and then we conclude by instantiating this tool from lattice assumptions.

Comparison. We compare the parameters we obtain in our scheme with the parameters obtained in the naive approach of RAM-to-circuit conversion and then applying previously known ABE for circuits schemes. While the main construction presented in the technical section has decryption complexity proportional to the worst case running time of the RAM programs, we can transform this scheme into another scheme where the decryption complexity is input-specific. This is performed by running \(\log(T)\) copies of the scheme by setting the worst case runtime of the first scheme to be 2, second scheme to be \(2^2\), so on and the runtime of the \(\log(T)\)\(^{th}\) scheme is set to be \(T\). This idea has been used in prior works (for instance, \cite{GKP+13b}). Note that this increases the size of the public parameters, keys and ciphertexts by a multiplicative factor of \(\log(T)\).

<table>
<thead>
<tr>
<th>Schemes</th>
<th>Size of Public key</th>
<th>Size of Ciphertext</th>
<th>Size of Key of RAM (P)</th>
<th>Decryption complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Via ABE for circuits \cite{BGG+14}</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot NT))</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot NT))</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot T))</td>
<td>(\tilde{O}((T + N)^3 \cdot n(\lambda,T)^2 \cdot T))</td>
</tr>
<tr>
<td>Via ABE for circuits \cite{BV16}</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot T))</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot NT))</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot T))</td>
<td>(\tilde{O}((T + N)^3 \cdot n(\lambda,T)^2 \cdot T))</td>
</tr>
<tr>
<td>Our Work</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot NT))</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot NT))</td>
<td>(\tilde{O}(n(\lambda,T)^2 \cdot N \cdot T^3))</td>
<td>(\tilde{O}(n(\lambda,t)^2 \cdot t))</td>
</tr>
</tbody>
</table>

Figure 1: We compare the parameters in our work with previous works. The polynomial \(n = n(\cdot,\cdot)\) denotes the lattice dimension and we set the polynomial \(m = \Theta(n \cdot \log(q))\) with \(q\) being the modulus. In both our work and previous works, we set \(q\) to be exponential in \(T\). The \(\tilde{O}\) notation suppresses poly-logarithmic factors (in \(N\) and \(T\)). The encryptor takes as input a database \(D\) of size \(N\). The attribute key is generated for a RAM program \(P\) with worst case runtime to be \(T\) and it takes time \(t\) to compute on \(D\). In previous works, an attribute key for \(P\) is generated by first transforming it into a circuit of size \((T + N)^3\) and depth \(T\) and then generating an attribute key for the resulting circuit. We omit comparison with Goldwasser et al. \cite{GKP+13b} since, as mentioned earlier, their modeling of ABE definition differs from us.

\(^2\)All the current known lattice-based ABE for general circuits \cite{BGG+14} are based on the same assumption.
In our scheme, in addition to the decryption complexity, the rest of the parameters in our system also depend on the upper bound on the running time. In contrast, the scheme of Goldwasser et al. [GKP+13b] achieve succinctness property, meaning that the encryption complexity and the key generation complexity is independent of the time bound. A natural question to ask here is whether we can achieve succinctness property without resorting to stronger assumptions. It turns out that an attribute based encryption satisfying succinctness property would imply succinct randomized encodings. This is because, attribute based encryption for RAMs satisfying succinctness implies succinct randomized encodings for Turing machines [BGJ+16, AJS15]. Current constructions of succinct randomized encodings are based on indistinguishability obfuscation [CHJV15, BGL+15, KLW15] for circuits.

Yet another point to note is that the key sizes in our scheme grows with the attribute length. Currently, we don’t have any evidence to suggest that this is inherent in LWE-based constructions and so it would be interesting to remove this restriction.

1.2 Technical Overview

We first present a brief description of one of the ABE for circuits schemes and in particular, we describe the scheme of Boneh et al. [BGG+14]. The reason why we choose [BGG+14] is that the construction is relatively easier to describe (as opposed to the construction of [GVW15a]) and also, some of the proof ideas introduced in [BGG+14] will be useful for us later.

Primer on [BGG+14]. The scheme is described as follows.

- The public key consists of matrices \( A, A_1, \ldots, A_N \in \mathbb{Z}_q^{n \times m} \) and the master secret key is a matrix \( T_A \in \mathbb{Z}_q^{m \times m} \) such that \( A \cdot T_A = 0 \).

- The encryption of an attribute \( D = (x_1, \ldots, x_N) \) and message \( \mu \) produces the ciphertext consisting of,
\[
T_A s + e^T \begin{pmatrix} T(A_1 + x_1 G) + e_1^T & \ldots & T(A_N + x_N G) + e_N^T \end{pmatrix}, \text{ Enc}(sk, \mu)
\]
where \( s \in \mathbb{Z}_q^n \) is a randomly chosen secret vector, \( G \) is the gadget matrix [MP12], \( e, \{e_i\} \) are error vectors (chosen from an appropriate Gaussian distribution) and \( \text{Enc} \) is a symmetric encryption scheme\(^4\) that allows for decrypting using “noisy” keys. In particular, given \( sk + err \), where \( err \) has small norm, we can distinguish \( \text{Enc}(sk, 0) \) and \( \text{Enc}(sk, 1) \).

- An attribute key corresponding to a circuit \( C \) is computed as follows: first homomorphically evaluate \( C \) on the matrices \( A_1, \ldots, A_\ell \) to obtain the matrix \( A_C \). The attribute key consists of the trapdoor \( T_C \), generated using the master secret key \( T_A \), such that the following holds: \( [A|A_C] \cdot T_C = sk \). We can view \( T_C \) as a recoding key; it recodes an encoding of a value under \( A_C \) into an encryption of the same value w.r.t \( \text{Enc} \) under \( sk \). If instead we need to compute an attribute key corresponding to a RAM program \( P \), we could first transform \( P \) into a circuit \( C \) and then compute an attribute key for \( C \) using the above procedure.

- The decryption consists of two steps: (i) homomorphism step: in this step, evaluate the ciphertexts \( \{T(A_i + x_i G) + e_i^T\} \) to obtain the ciphertext that is approximately \( T(A_C + \ldots
\( C(D)G \), (ii) **authentication step**: in this step, we use the homomorphically computed ciphertext and the trapdoor \( T_C \) to obtain a noisy secret key \( sk \) only if \( C(D) = 0 \). The noisy key then allows us to obtain the message \( \mu \).

At a high level, the security proof proceeds in the following steps: let the challenge attribute chosen by the adversary be \( x^* \) and attribute keys are generated for circuits \( C_1, \ldots, C_q \) such that \( C_i(x^*) = 1 \). To show that the scheme is secure, we need to argue that the challenge ciphertext and the attribute keys can be simulated.

- **Simulation of Public Keys**: The \( i^{th} \) public key \( A_i \) is simulated using the \( i^{th} \) bit of \( x^* \).
- **Simulation of Recoding Key**: Suppose we want to simulate an attribute key for a circuit \( C \). Since the matrices \( \{A_i\}_{i \in [N]} \) are simulated, the result of homomorphically evaluating the matrices \( \{A_i\}_{i \in [N]} \) using \( C \) results in a matrix \( A_C \) that is simulated using \( C(x^*) \). This fact is crucially used to simulate the trapdoor \( T_C \) and in particular, the trapdoor \( T_A \) is not used to generate \( T_C \).
- **Pseudorandomness of Ciphertexts**: At this point, the ciphertext is sampled from uniform distribution instead of generating it as \( \text{Enc}(sk, \mu) \). A computationally bounded adversary will not be able to distinguish this switch from the learning with errors assumption.

**Constructing ABE for RAMs: First Attempt.** We first discuss the hurdles involved in extending the scheme of Boneh et al. [BGG+14] to directly construct ABE for RAMs and in particular avoiding the costly RAM-to-circuit conversion.

Notice that the attribute key \( T_C \) in [BGG+14] is generated as a function of the matrices \( \{A_i\} \) and circuit \( C \). This means key generation algorithm knows all the operations, specified by the circuit \( C \), that is to be performed on the data and thus can authenticate only those operations that are legal. As an example, consider a circuit that consists of applying OR gates to its input and then applying a giant AND gate at the top. At the time of generating the key for this circuit, the authority knows that first applying OR and then AND is the only legal computation path that can be taken and it can thus generate a trapdoor that only authenticates this computation. However, if we were to generate attribute keys for RAM programs directly then we would run into trouble. The operations performed during RAM computation can be highly data-dependent (unlike circuits, which consist of data-oblivious operations) and hence it is unclear which set of operations to authenticate during the key generation process. For instance, a RAM program \( P \) could read the first bit of the database and if its value is 0 it executes a sequence of OR gates and then applies a giant AND gate, otherwise if its value is 1 then it could simply output the second bit of the database. This means that the computation path, i.e., a sequence of operations to be performed on the data, is ill-defined during the key generation phase and hence its unclear how to execute the authentication mechanism.

A first attempt to solve the above issue is enumerate all possible computation paths and then generate a trapdoor for every computation path. In more detail, let \( T \) be an upper bound on the running time of the program and for now, think of \( T \) as being a constant. This means that all possible \( T \)-sized subsets of the memory locations can be accessed by the program during decryption. For every possible \( T \)-sized set \( I \subseteq [N] \), we first perform homomorphic evaluation on the matrices \( \{A_i\}_{i \in I} \) to obtain the matrix \( A_I \). The next step is to generate a trapdoor \( T_I \) such that \( [A]_I \cdot T_I = sk \). Since \( T \) is a constant, the size of the attribute key is polynomial sized, as desired. On input an encryption of attribute \( D \) and message \( \mu \), first determine the set of locations \( I^* \) accessed by the program. Then use the trapdoor \( T_{I^*} \) to obtain the noisy key and decrypt the message as before.
This scheme achieves sublinear decryption complexity: the decryption algorithm only needs to touch ciphertext encodings computed with respect to \( \{A_i\}_{i \in I^*} \) and trapdoor \( T_{I^*} \). However, in terms of security, this scheme fails. There is no mechanism in place that prevents a malicious evaluator from illegally using a trapdoor \( T_{I'} \), for \( I' \neq I^* \). This suggests that we need a controlled authentication mechanism that lets us evaluate only “legal” trapdoors depending on the data. Moreover, even if we tweak the scheme to incorporate this mechanism, a bigger problem is that this does not scale for the case when \( T \) is not a constant since the attribute key would no longer be polynomial sized. We introduce the notion of controlled homomorphic recoding schemes that overcomes the above barriers.

**Our Approach: Controlled Homomorphic Recoding Scheme.** The main insight in our approach is to divide the computation into several tiny modules of computation and then apply authentication mechanism after the execution of every module. A RAM program presents a natural way to achieve such a modularization: a module corresponds to the associated step circuit of the RAM program. As in the case of [BGG+14], the encryption will consist of encodings of the database. We design the decryption process to proceed by homomorphically evaluating the step circuit followed by authenticating its output which then is followed by homomorphic evaluation of the step circuit for next time step and so on. In order to perform authentication after every time step, we provide \( T \) auxiliary keys as part of the attribute key, where \( T \) is the maximum running time of the associated RAM program. The main challenge we face when we try to nail down this approach is that we need a mechanism to ‘stitch’ the intermediate homomorphism and the authentication steps together. Specifically the authentication phase should not only verify the correctness of the computation of the step circuit but it should also pass along the valid encoded information to the next homomorphism phase. We term this phase, that performs the job of both authentication and translation of encodings, as controlled recoding phase. Incorporating both the homomorphism phase and the controlled recoding phase, we introduce the notion of controlled homomorphic recoding scheme.

A controlled homomorphic recoding scheme allows for encoding messages \( x \) along with secret randomness \( s \) with respect to public key \( pk \). There are two main phases associated with a controlled homomorphic recoding scheme, namely, (1) public homomorphism phase and, (2) controlled recoding phase.

In more detail, we describe the algorithms associated with a controlled homomorphic recoding scheme. Setup generates the public key \( pk \) and secret key \( sk \). Enc is a mechanism to transform attribute \( y \) and secret message \( s \) into a ciphertext \( ct \). Equality test \( \text{EqTest} \) allows for checking if two different ciphertexts \( ct_1 \) and \( ct_2 \) encode the same attribute bit, given the condition that they both are computed with respect to the same public key \( pk \) and the same secret message \( s \). The rest of the algorithms are classified into public homomorphism and controlled recoding phases.

**Public Homomorphism:** There are two algorithms associated with this phase. The first algorithm \( \text{KeyEval} \) takes as input many public keys \( pk_1, \ldots, pk_n \), circuit \( C \) and outputs a homomorphically evaluated public key \( pk_C \). The second algorithm, takes as input ciphertexts \( ct_1, \ldots, ct_n \) with \( ct_i \) computed under \( pk_{j_i} \), circuit \( C \) and it computes the ciphertext \( ct^* \) under the resulting public key \( C(pk_1, \ldots, pk_n) \). Looking ahead, \( C \) will essentially represent the step circuit of a RAM program. We present a pictorial representation of both these algorithms in Figure 3.

**Controlled Recoding:** The main goal of this phase is two-fold: first verify the computation in the previous time step and if the verification phase succeeds then produce encodings for the
Figure 2: A high level description of how the two phases (homomorphism and controlled recoding) mirror the execution of the RAM program. $x_i$ denotes the encoding of $x_i$. The controlled recoding phase translates the encodings of state and "read $j$" instruction from the previous time step to the next time step. It also translates the $j^{th}$ database encoding into an encoding for the next time step.

Figure 3: Description of homomorphism algorithms. The topmost figure denotes the execution of KeyEval and the next figure denotes the execution of CtEval.
next homomorphism phase. The verification step implicitly captures the controlled authentication mechanism that we touched upon earlier.

There are two algorithms associated with the controlled recoding phase. The ciphertext recoding procedure \texttt{ReEnc} allows for recoding ciphertexts of \( \{x_i\}_{i \in [n]} \) under public keys \((pk_1, \ldots, pk_n)\) into a ciphertext of \( f(x_1, \ldots, x_n) \) under the public key \( pk^* \) as long as \( f(x_1, \ldots, x_n) \neq \perp \). This recoding process is carried out with the help of a recoding key \( rk_f \), which is associated with a control function \( f \). The recoding key generation algorithm \texttt{ReEncKG} allows for generating such a recoding key, \( rk_f \). Looking ahead, in the construction of ABE from CHR, the control functions will be critical in controlling the information to be passed on from the output of step circuit in the \( i^{th} \) step to \( (i+1)^{th} \) step.

Consider the following example. Let \( rk_f \) be a recoding key that recodes ciphertexts under public keys \((pk_1, pk_{100})\), where \( f \) is a function that takes as input \((x, y)\) and outputs \( y \) if \( x = 100 \), otherwise it outputs \( \perp \). This is useful for the reading operation in the ABE application. We can think of the public key \( pk_1 \) being used to encode the read address 100 and \( pk_{100} \) used to encode the value in the 100th memory location.

We explain the correctness requirement by considering a toy example. Consider three input bits \((x_1, x_2, x_3)\), circuits \( C_1, C_2 \), and control function \( f \) such that \( f(C_1(x_1, x_2, x_3), C_2(x_1, x_2, x_3)) \in \{0, 1\} \).

- Suppose \( ct_1, ct_2, ct_3 \) are encodings of \((x_1, x_2, x_3)\) (under the same randomness) respectively under the public keys \((pk_1, pk_2, pk_3)\).
- Homomorphically evaluating \((ct_1, ct_2, ct_3)\) using the circuit \( C_1 \) (resp., \( C_2 \)) yields ciphertext of \( C_1(x_1, x_2, x_3) \) (resp., \( C_2(x_1, x_2, x_3) \)) under the public key \( C_1(pk_1, pk_2, pk_3) \) (resp., \( C_2(pk_1, pk_2, pk_3) \)). Call these two encodings \( ct'_1 \) and \( ct'_2 \).
- Suppose \( rk_f \) is a recoding key that translates ciphertexts encoded under the public keys \((pk_1, pk_2, pk_3)\) into a ciphertext under public key \( pk^* \). Upon executing \texttt{ReEncKG} on input \( ct'_1, ct'_2 \) and recoding key \( rk_f \), let \( ct^* \) be the resulting ciphertext.

We require the following condition to hold: \texttt{EqTest} should declare \( ct^* \) and \( ct \) equal, as long as \( ct \) is a ciphertext of \( f(C_1(x_1, x_2, x_3), C_2(x_1, x_2, x_3)) \) and secret randomness \( s \) under the public key \( pk^* \).

**Main Construction: ABE for RAMs from CHR.** We now show how to construct ABE for RAMs starting from a controlled recoding scheme. We only provide a high level template below and this suffices to understand the main ideas in our construction. We also later identify some technical challenges that arise when we try to implement this template and how to handle them.
**SETUP:** Let $N$ be the length of the attribute. There are three main types of CHR public keys to be generated: (i) public keys ($\text{Step}[0].\text{pk}_{db}^0, \ldots, \text{Step}[0].\text{pk}_{db}^N$) corresponding to the attribute database, (ii) anchor public key-secret key pair ($\text{pk}_{0}, \text{sk}_0$), (iii) target public key $\text{pk}_{out}$. In addition, CHR public key used to encode the initial read address, namely $\text{Step}[0].\text{pk}_{ta}^0$ and the CHR public key used to encode the initial state information $\text{Step}[0].\text{pk}_{st}^0$. All these public keys, denoted by $\text{Step}[0].PK$, will be part of the ABE public parameters.

The anchor secret key $\text{sk}_0$ will be set as the master secret key of the ABE scheme.

**KEY GENERATION:** Let $P$ be the RAM program for which we need to generate the ABE key with run time upper bounded by $T$ and let $C$ be the step circuit associated with $P$.

Sample public keys for every step in $[T - 1]$ and the number of such public keys for every step is proportional to the input length of $C$. That is, generate $\text{Step}[1].PK, \ldots, \text{Step}[T - 1].PK$, where $\text{Step}[i].PK$ denotes the set of public keys associated with the $i$-th step.

The next step is to generate recoding keys $\text{Step}[1].RK, \ldots, \text{Step}[T - 1].RK$, where the recoding keys in the set $\text{Step}[i]$ recodes the encodings w.r.t the $\text{Step}[i - 1]$ public keys to encodings w.r.t the $\text{Step}[i]$ public keys.

Execute the following two steps for every time step $t \in [T]$:

- **HOMOMORPHISM KEYS:** Execute the key evaluation algorithm $\text{KeyEval}$ of CHR on the set of public keys on $\text{Step}[t - 1].PK$ to obtain the set of public keys $\text{Step}[t - 1].PK_{\text{hom}}$. The public keys in $\text{Step}[t - 1].PK_{\text{hom}}$ is used to encode the output of $C$ in the $(t - 1)$-th step.

- **CONTROLLED RECODING KEYS:** Execute the key recoding algorithm $\text{ReEncKG}$ of CHR on the public keys $\text{Step}[t - 1].PK_{\text{hom}}$ and control functions in the class $F$ to obtain the set of recoding keys $\text{Step}[t].RK$, for every time step $t \in [T]$. The class $F$ is used to translate the output of the $(t - 1)$-th step circuit to the input of $t$-th step circuit.

To give a glimpse of what $F$ contains, we give two examples:

- $\text{Ind}$: this is an identity function. This is useful in converting an encoding of state output by the previous step into an encoding input to the next step. This is also useful in transferring the read address output by previous step to the next one.

- $f_i(i', b)$: this outputs $b$ only if $i = i'$. This is useful for writing operation: suppose the step circuit at some time $t$ outputs a location $i'$ and value $b$ to be written to. In this case, a recoding key associated with $f_i$ will transform encoding of location $i'$ into encoding of $b$ in the $i$-th database location only if $i = i'$.

Set the ABE key of the program $P$ to be ($\text{Step}[1].RK, \ldots, \text{Step}[T].RK$).

**ENCRYPTION:** It takes as input attribute $x$ of size $N$ and message $\mu$. As before, it first samples secret randomness $s$ from a distribution. It computes the following encodings: (i) encoding of $x_i$ and $s$ under the public key $\text{pk}_i$, (ii) encoding of 0 and $s$ under the public key $\text{pk}_0$ and finally, (iii) encoding $\mu$ of $\mu$ and $s$ under the public key $\text{pk}_{out}$. Additionally, it also computes encoding of initial read address (set to 1) under $\text{Step}[0].\text{pk}_{ta}^0$ and the encoding of initial state (also set to 1s). All the encodings computed will be part of the ABE ciphertext.

**DECRYPTION:** This proceeds in $T$ steps, where $T$ is the runtime of the RAM program and in each step, it executes homomorphism and controlled recoding phases. In more detail, in the $t$-th step, it executes:
• **HOMOMORPHISM:** The step circuit $C$ is homomorphically evaluated on the encodings output by $(t - 1)$-th step to obtain encodings of output of $t$-th step under the public keys in $\text{Step}[t].PK^{\text{hom}}$.

• **CONTROLLED RECODING:** Using the recoding keys in $\text{Step}[t].RK$, the encodings computed under the public keys in $\text{Step}[t].PK^{\text{hom}}$ can be recoded to encodings computed under the public keys in $\text{Step}[t].PK$. As described earlier, the recoding keys determine what value to be fed to the $t$-th time step as a function of the $(t - 1)$-th time step.

In the last step, once we have an encoding of 0 and $s$ under $pk_{\text{out}}$, (as before) we then run the equality test on this encoding and $ct^*$ (as computed in encryption). If they are equal, this means that the secret message $\mu$ has to be 0, otherwise it has to be 1.

**Challenges.** The template described above captures the main ideas in our construction. However, while implementing this high level template, we encounter additional difficulties and we highlight a couple of them below.

**Repeated Writing Issue.** Yet another issue is that of malicious execution of the computation. Suppose the 100th location was updated in the 11th step and also in the 25th step. Lets consider what happens when the RAM program in the 30th step is supposed to read the 100th location. A malicious evaluator could use the encryption computed in the 11th step to be input to the 30th step, instead of 25th step. We need to implement suitable checks in place that prevents him from performing these types of attacks.

In the technical sections, we introduce circuits $C^{\text{up}}$ (Figure 5) and $C^{\text{ck}}$ (Figure 6) that keeps track of all the addresses written so far along with the along with the most recent time stamps associated with them. We also introduce the control function $f_{ij}$ (Figure 2) is used to ensure that only the correct encoding is recoded.

**Early Termination.** What if the program terminates much earlier than the upper time bound $T$? The template described so far, as is, would have the decryption algorithm run in $T$ steps even if the program terminated early. We solve this by giving out multiple keys for programs upper bounded by runtime $2, 2^2, \ldots, 2^{\log(T)}$. In particular, using this we can achieve input-specific runtime. This would introduce an additional overhead of $\log(T)$ in the size of the original key.

**Security Overview.** Ignoring the additional challenges for now, we describe the main steps in the security proof of the above sketched template. Before sketching the steps in the security proof, we give an overview of the security proof. As a simple example, let us consider the case when the adversary receives a single challenge ciphertext of $(D^*, \mu^*)$ and an attribute key of program $P$. The first step is to simulate the public key using the database $D^*$. Next, we simulate the attribute key of $P$. Recall that the attribute key consists of $T$ layers of recoding keys, one for every step in the computation of $P$. Using the simulation of the public keys, the first layer recoding keys are simulated. This is then propagated to simulation of second layer recoding keys and so on. We explicitly describe below the security properties needed to carry out this simulation process.

**Step 1. Simulation of Public Keys.** As in the case of ABE for circuits scheme, the first step is to simulate the public keys produced by $\text{Sim.CHRSsetup}$. In particular, the challenge attribute $D^*$ is programmed in $\text{Sim.CHRSsetup}$. To carry out this step, we need to define a security property
for the underlying CHR scheme.

I. Indistinguishability of Setup: To define this property, we first define a simulator $\text{Sim.CHRSsetup}$ that takes as input a value $v$ to be programmed and outputs a public key $\text{Sim.pk}$ and a secret trapdoor $\tau$. We require that the distribution of simulated public keys $\{\text{Sim.pk}\}$ is indistinguishable to the distribution of real public keys $\{pk\}$.

**Step 2. Simulation of Intermediate Recoding Keys.** The next goal is to simulate the intermediate recoding keys (i.e., $\text{Step[1].RK}, \ldots, \text{Step[T-1].RK}$) in every attribute key. In particular, these recoding keys need to be generated without the help of the anchor secret key. Recall that in the case of ABE for circuits scheme, we could simulate the recoding key $rk_f$ since the output of $f$ on the challenge attribute was guaranteed to be 1. However, in the setting of ABE for RAMs, we have no such guarantee for the intermediate steps of the computation. In particular, there could two programs $P_0$ and $P_1$ that output 1 on $x^*$ but differ on every intermediate step of the computation. Thus, we can no longer invoke the indistinguishability of the recoding keys property.

To handle this case, we introduce the following security property associated with the CHR scheme.

II. Indistinguishability of Simulated Keys: We first define an associated simulator $\text{Sim.CHRSkey}$. In its basic form, it takes as input anchor public key $pk$, simulated public keys ($\text{Sim.CHRSpk}_1, \ldots, \text{Sim.CHRSpk}_n$), associated trapdoors ($\tau_1, \ldots, \tau_n$), homomorphism circuit $C$, control function $f$ and it produces simulated recoding keys associated with $(C, f)$ along with simulated target public keys.

Let's see how to use the above security property to simulate the intermediate recoding keys in the attribute keys. For simplicity, consider the case when the adversary only makes a single attribute key query for RAM program $P$. Using a standard hybrid argument, we can apply the argument for the case of multiple key queries as well. As a first step, we switch the recoding keys $\text{Step[1].RK}$ in the attribute key of $P$ to simulated recoding keys using the above security property. Note that even the intermediate public keys $\text{Step[1].PK}$ are simulated. In particular, we use the fact that the public keys in $\text{Step[0].PK}$ are simulated using $\text{Sim.CHRSsetup}$. Next, we simulate the recoding keys in $\text{Step[2].RK}$. Recall that $\text{Step[2].RK}$ was computed as a function of $\text{Step[1].PK}$ and the step circuit associated with $P$. Hence, in order to simulate $\text{Step[2].RK}$ we first need to simulate $\text{Step[1].PK}$. But note that we already simulated $\text{Step[1].PK}$ by $\text{Sim.CHRSkey}$ in the previous step itself! This allows for carry out successful simulation of $\text{Step[2].RK}$.

A remark about the definition of indistinguishability of simulated keys: there are two ways to generate the simulated public keys ($\text{Sim.CHRSpk}_1, \ldots, \text{Sim.CHRSpk}_n$). We can use $\text{Sim.CHRSsetup}$ to generate these keys. Indeed, to argue the security of the recoding keys in $\text{Step[1].RK}$, the public keys in $\text{Step[0].PK}$ is simulated using $\text{Sim.CHRSsetup}$. Another option is to invoke $\text{Sim.CHRSkey}$ to generate the $\{\text{Sim.CHRSpk}_i\}$. This is not circular since the simulated public keys produced by $\text{Sim.CHRSkey}$ in the first step is used in the second step by $\text{Sim.CHRSkey}$ to produce the recoding keys in $\text{Step[2].RK}$. In the technical sections, we formalize this by associating a distribution $\mathcal{E}_{aux}$ which produces $\{\text{Sim.CHRSpk}_i\}$.

---

\[5\text{We emphasize that the intermediate public keys are generated afresh for every attribute key. This enables us to apply the hybrid argument for the case of multiple key queries.}\]
Step 3. Simulation of Final Step Recoding Keys. Continuing this way, we can simulate all the recoding keys in Step[1].RK,...,Step[T - 1].RK. We cannot, however, use the indistinguishability of simulated keys property to simulate Step[T].RK. This is because, the simulator Sim.CHR_key would end up simulating the target key, which in our construction is pk^out. In turn this means that we cannot apply the indistinguishability of simulated keys property (for the multiple key queries case) as pk^out is reused across different attribute keys. To get round this, we define the indistinguishability of recoding keys property below.

III. Indistinguishability of Recoding Keys: The simulator Sim.CHR_rk is defined as follows: Let the public keys used to encode the database be simulated as (Sim.pk_1,...,Sim.pk_n), where the i^{th} attribute bit x_i is programmed in Sim.pk_i. As long as the output of C on x is not 0, the recoding key rk_f associated with the attribute key of C, can be generated without the secret key sk_0.

The indistinguishability of recoding keys property states that the distribution of honestly generated recoding keys is indistinguishable from the distribution of simulated recoding keys.

Step 4. Simulate Encryption of Secret Message. Once the anchor secret key is not used in the generation of the recoding keys for any of the attribute keys, we can now invoke the pseudorandomness of ciphertexts property (defined below) to argue that the secret message in the ABE encryption is hidden. This completes the security proof.

IV. Pseudorandomness of Ciphertexts: This property states that the encoding of b and secret randomness s, under a public key pk, is indistinguishable from uniform distribution on the space of encodings.

Instantiation of CHR. It remains to show that the controlled homomorphic recoding schemes can be based on learning with errors. We first observe that it suffices to define a controlled homomorphic recoding schemes only for a specific class of control functions. See below for a description of the control functions and we explain in Table 1 how they are used in the ABE construction.

Here, N denotes the length of the attribute and T denotes the maximum running time of the RAM programs.

$$\mathcal{F} = \{\text{Ind}, h, \{g_i\}, \{f_{ij}\}_{i \in [N]}, \{f_{ij}\}_{i \in [N], j \in [T]}\}$$

(1)

where,

$$f_{ij}(i', j', b) = \begin{cases} 
  b, & \text{if } i = i' \land j = j' \\
  \bot, & \text{otherwise}
\end{cases} \quad f_i(i', b) = \begin{cases} 
  b, & \text{if } i = i' \\
  \bot, & \text{otherwise}
\end{cases}$$

(2)

$$g_i(x) = \begin{cases} 
  i, & \text{if } C(x) = i - 1 \\
  \bot, & \text{otherwise}
\end{cases} \quad h(x) = \begin{cases} 
  1, & \text{if } x = 0 \\
  \bot, & \text{otherwise}
\end{cases}$$

(3)

where C is a gadget circuit defined as \(C(x) = \sum_{i=1}^{L} x_i 2^i\), and \(x = (x_1, \ldots, x_L)\). Finally we define the identity function, \(\text{Ind}(x) = x\).

The template for encoding and the key generation is inspired by the schemes of Gorbunov et al. [GVW15a] and Boneh et al. [BGG+14]. To encode a message b with secret randomness s under the public key pk, our encoding is of the form \(s^T(A + bG) + e^T\), where \(s^T, A\) and e are sampled according to the parameters associated with the learning with errors assumption. Suppose we have
<table>
<thead>
<tr>
<th>Functions</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Ind}</td>
<td>recode current state (reading address) to next step</td>
</tr>
<tr>
<td>\text{f}_{ij}</td>
<td>recode value of current reading address to next step</td>
</tr>
<tr>
<td>\text{f}_i</td>
<td>recode writing value to current writing address</td>
</tr>
<tr>
<td>\text{g}_i</td>
<td>recode ((i-1))-th time step to (i)-th time step</td>
</tr>
<tr>
<td>\text{h}</td>
<td>recode current step to final step if the program terminates at this step</td>
</tr>
</tbody>
</table>

Table 1: Usage of Controlled Functions in ABE Setting

many encodings \( s^T (A_1 + b_1 G) + e_1^T, \ldots, s^T (A_n + b_n G) + e_n^T \) then we can compute an encoding of the form \( s^T (A_C + C(b_1, \ldots, b_n) G) + e^T \), where \( A_C \) is homomorphically computed on public keys \( A_1, \ldots, A_n \).

To handle the recoding process, we need to generate recoding keys individually for every control function described above. The recoding keys are set to be lattice trapdoors. As an illustration, we show how to generate lattice trapdoor for the case of control function \text{Ind}.

- \text{Ind}: Suppose the input to the recoding key generation is anchor public key \( pk_0 \), secret key \( sk_0 \), public key \( pk_1 \), target public key \( pk_{tgt} \) and control function \text{Ind}. We set \( pk_0 = A_0 \), \( sk_0 = T_{A_0} \) (a trapdoor for \( A_0 \)), public key \( pk_1 = A_1 \), \( pk_{tgt} = A_{tgt} \). The recoding key is of the form \( [R_0 | I]^T \) such that \( [A_0 | A_1] \cdot [R_0 | I]^T = A_{tgt} \). Using this recoding key, we can translate encoding of any message \( b \) under \( A_1 \) into an encoding of \( b \) under \( A_{tgt} \).

We use similar ideas to generate the other recoding keys for the control functions that are relevant to the construction of ABE for RAMs.

1.3 Related Work

The constructions of ABE systems has a rich literature. The seminal result of Goyal, Pandey, Sahai and Waters [GPSW06] presented the first construction of ABE for boolean formulas from bilinear DDH assumption. Since then, several prominent works achieved stronger security guarantees [LOS\textsuperscript{+}10], better efficiency or design guarantees [Wee14, Att14, AC16] and achieving stronger models of ABE for a restricted class of functions [KSW08]. The breakthrough work of Gorbunov, Vaikuntanathan and Wee [GVW15a] presented the first construction of ABE for all polynomial-sized circuits assuming learning with errors. Following this, several works [BGG\textsuperscript{+}14, BV16] improved this result in terms of efficiency and also considering stronger security models [GVW15a]. In addition to [GKP\textsuperscript{+}13b], there are a few works that consider ABE in other models of computation. Waters [Wat12] proposed a construction of functional encryption for regular languages and subsequently Agarwal and Singh [AS17] constructed reusable garbled finite automata from LWE. Ananth and Sahai [AS16] construct functional encryption for Turing machines assuming sub-exponentially secure functional encryption for circuits. Deshpande et al. [DKW16] present an alternate construction of attribute based encryption for Turing machines under the same assumptions.

2 Preliminaries

\textbf{Notation.} Let \( \lambda \) denote the security parameter, and \textsc{ppt} denote probabilistic polynomial time. Bold uppercase letters are used to denote matrices \( M \), and bold lowercase letters for vectors \( v \). We use \([n]\) to denote the set \( \{1, \ldots, n\} \). We say a function \( \text{negl}(\cdot): \mathbb{N} \rightarrow (0, 1) \) is negligible, if for every
constant \( c \in \mathbb{N} \), \( \negl(n) < n^{-c} \) for sufficiently large \( n \). Let \( X \) and \( Y \) be two random variables taking values in \( \Omega \). Define the statistical distance, denoted as \( \Delta(X, Y) \) as

\[
\Delta(X, Y) := \frac{1}{2} \sum_{s \in \Omega} |\Pr[X = s] - \Pr[Y = s]|
\]

Let \( X(\lambda) \) and \( Y(\lambda) \) be distributions of random variables. We say that \( X \) and \( Y \) are statistically close, denoted as \( X \approx Y \), if \( d(\lambda) := \Delta(X(\lambda), Y(\lambda)) \) is a negligible function of \( \lambda \). We say two distributions \( X(\lambda) \) and \( Y(\lambda) \) are computationally indistinguishable, denoted as \( X \approx Y \) if for any ppt distinguisher \( D \), it holds that \( |\Pr[D(X(\lambda)) = 1] - \Pr[D(Y(\lambda)) = 1]| = \negl(\lambda) \).

### 2.1 Random Access Machines

We recall the definition of RAM program in [GHL+14]. A RAM computation consists of a RAM program \( P \) and a database \( D \). The representation size of \( P \) is independent of the length of the database \( D \). \( P \) has random access to the database \( D \) and we represent this as \( P^D \). On input \( x \), \( P^D(x) \) outputs the answer \( y \). In more detail, the computation proceeds as follows.

The RAM program \( P \) is represented as a step-circuit \( C \). It takes as input internal state from the previous step, location to be read, value at that location and it outputs the new state, location to be written into, value to be written and the next location to be read. More formally, for every \( i \in T \), where \( T \) is the upper running time bound

\[
(st_i, loc^w_i, b^w_i, loc^r_i) \leftarrow C(st_{i-1}, loc^r_{i-1}, b^r_{i-1}),
\]

where we have the following:

- \( st_{i-1} \) denotes the state from the \((i - 1)\)-th step and \( st_i \) denotes the state in the \(i\)-th step. Initial state \( st_0 \) is set to be \( x \), which is the input to \( P^D(\cdot) \).
- \( loc^r_{i-1} \) denotes the location to be read from, as output by the \((i - 1)\)-th step.
- \( b^r_{i-1} \) denotes the bit at the location \( loc^r_{i-1} \).
- \( loc^r_i \) denotes the location to be read from, in the next step.
- \( loc^w_i \) denotes the location to be written into.
- \( b^w_i \) denotes the value to be written at the location \( loc^w_i \).

At the end of the computation, denote the final state to be \( st_{\text{end}} \). If the computation has been performed correctly, \( st_{\text{end}} = y \). In this work, we consider a simpler case, where the RAM program \( P \) does not take additional input \( x \) and the output of \( P^D \) is in space \( \{0, 1\} \).

### 2.2 Attribute-Based Encryption for RAMs

In this part, we recall the syntax and security definition of (key-policy) attribute-based encryption (ABE). An ABE scheme for a RAM program \( P \) and a database \( D \) consists a tuple of ppt algorithms \( \Pi = (\text{Setup}, \text{KeyGen}, \text{Enc}, \text{Dec}) \) with details as follows:

- **Setup**, \( \text{Setup}(1^\lambda, 1^T) \): On input security parameter \( \lambda \) and upper time bound \( T \), setup algorithm outputs public parameters \( pp \) and master secret key \( msk \).
• **Key Generation**, $\text{KeyGen}(\text{msk}, P)$: On input a master secret key $\text{msk}$ and a RAM program $P$, it outputs a secret key $\text{sk}_P$.

• **Encryption**, $\text{Enc}(\text{pp}, D, \mu)$: On input public parameters $\text{pp}$, a database $D$ and a message $\mu$, it outputs a ciphertext $\text{ct}_D$.

• **Decryption**, $\text{Dec}(\text{sk}_P, \text{ct}_D)$: This is modeled as a RAM program. In particular, this algorithm will have random access to the binary representations of the key $\text{sk}_P$ and the ciphertext $\text{ct}_D$. It outputs the corresponding plaintext $\mu$ if the decryption is successful; otherwise, it outputs ⊥.

We define the properties associated with the above scheme next.

**Definition 2.1** (Correctness). We say the ABE described above is correct, if for any message $\mu$, any RAM program $P$, and any database $D$ where $P^D$ outputs 0, we have $\text{Dec}(\text{sk}_P, \text{ct}_D) = \mu$, where $(\text{msk}, \text{pp}) \leftarrow \text{Setup}(1^\lambda, 1^T)$, $\text{sk}_P \leftarrow \text{KeyGen}(\text{msk}, P)$ and $\text{ct}_D \leftarrow \text{Enc}(\text{pp}, D, \mu)$.

**Efficiency.** We define two efficiency properties associated with a ABE for RAMs scheme: namely sub-linear decryption and input-specific runtime property. The latter property implies the former.

**Sub-linear Decryption:** This property states that the complexity of decryption is $p(\lambda, T)$ for some fixed polynomial $p$. We call this sub-linear decryption for the following reason: suppose $T$ is sufficiently sublinear in $|D|$ (for instance, poly-logarithmic in $|D|$) then the decryption time is sub-linear in $|D|$. More specifically, suppose $p(\lambda, T) = \lambda^{c'} \cdot T^c$ and if $T << |D|^{\frac{1}{c}}$ then the decryption complexity is sub-linear in $|D|$.

**Definition 2.2** (Sublinear Decryption). An ABE for RAMs scheme $\text{ABE}$ is said to satisfy sub-linear decryption property if the following holds: for any database $D$, message $\mu$, program $P$, (i) $(\text{msk}, \text{pp}) \leftarrow \text{Setup}(1^\lambda, 1^T)$, (ii) $\text{sk}_P \leftarrow \text{KeyGen}(\text{msk}, P)$ for some RAM program $P$, (iii) $\text{ct} \leftarrow \text{Enc}(\text{pp}, D, x)$ and, (iv) the decryption $\text{Dec}$ of the functional key $\text{sk}_P$ on input the ciphertext $\text{ct}$ takes time $\text{poly}(T, \lambda)$, where $T$ is the running time of $P^D$.

**Input-specific Runtime:** This property states that the time to decrypt a ciphertext $\text{ct}$ of $(D, \mu)$ using an attribute key of $\text{sk}_P$ is $p(\lambda, t)$ for some fixed polynomial $p$, where $t$ is the execution time of $P$ on input database $D$.

**Definition 2.3** (Input-specific Runtime). An ABE for RAMs scheme $\text{ABE}$ is said to satisfy input-specific runtime property if the following holds: for any database $D$, message $\mu$, program $P$, (i) $(\text{msk}, \text{pp}) \leftarrow \text{Setup}(1^\lambda, 1^T)$, (ii) $\text{sk}_P \leftarrow \text{KeyGen}(\text{msk}, P)$ for some RAM program $P$, (iii) $\text{ct} \leftarrow \text{Enc}(\text{pp}, D, x)$ and, (iv) the decryption $\text{Dec}$ of the functional key $\text{sk}_P$ on input the ciphertext $\text{ct}$ takes time $\text{poly}(t, \lambda)$, where $t$ is the running time of $P^D$.

**Remark 2.1.** While the above properties focus on the decryption complexity, we can also correspondingly define efficiency measures for setup, key generation and encryption. Since the focus of this work is on decryption complexity, we postpone the discussion of these properties to future works.

**Security Definition.** We present the simulation-based definition of selective security of attribute-based encryption as follows
Definition 2.4. An ABE scheme II for RAMs is simulation-based selectively secure if there exist PPT simulator $S = (S_1, S_2, S_3)$ such that for any PPT admissible adversary $A = (A_1, A_2)$, the two distributions \( \{ \text{Expt}^\text{real}(1^\lambda) \}_\lambda \approx \{ \text{Expt}^\text{ideal}(1^\lambda) \}_\lambda \) are computationally indistinguishable.

We call adversary $A = (A_1, A_2)$ admissible, if the queries $P_i$ made by $A_2$ satisfies $P_i(D^*) \neq 0$.

Remark 2.2. We note that we can generalize the ABE syntax, by allowing RAM program $P$ to take in auxiliary input $x$, denoted as $P^D(x)$. The encryption algorithm $\text{Enc}(pp, D, x, \mu)$ outputs ciphertext $ct_{D,x}$ associated with database $D$ and auxiliary input $x$. Correctness and security can be defined similarly by replacing database $D$ with $(D, x)$.

2.3 Lattice Background

A full-rank $m$-dimensional integer lattice $\Lambda \subset \mathbb{Z}^m$ is a discrete additive subgroup whose linear span is $\mathbb{R}^m$. The basis of $\Lambda$ is a linearly independent set of vectors whose linear combinations are exactly $\Lambda$. Every integer lattice is generated as the $\mathbb{Z}$-linear combination of linearly independent vectors $B = \{b_1, ..., b_m\} \subset \mathbb{Z}^m$. For a matrix $A \in \mathbb{Z}_q^{n \times m}$, we define the “$q$-ary” integer lattices:

\[
\Lambda_q^\perp = \{ e \in \mathbb{Z}^m | A e = 0 \mod q \}, \quad \Lambda_q^u = \{ e \in \mathbb{Z}^m | A e = u \mod q \}
\]

It is obvious that $\Lambda_q^u$ is a coset of $\Lambda_q^\perp$.

Let $\Lambda$ be a discrete subset of $\mathbb{Z}^m$. For any vector $c \in \mathbb{R}^m$, and any positive parameter $\sigma \in \mathbb{R}$, let $\rho_{\sigma,c}(x) = \exp(-\pi||x - c||^2/\sigma^2)$ be the Gaussian function on $\mathbb{R}^m$ with center $c$ and parameter $\sigma$. Next, we let $\rho_{\sigma,c}(\Lambda) = \sum_{x \in \Lambda} \rho_{\sigma,c}(x)$ be the discrete integral of $\rho_{\sigma,c}$ over $\Lambda$, and let $D_{\Lambda,\sigma,c}(y) := \rho_{\sigma,c}(y)/\rho_{\sigma,c}(\Lambda)$. We abbreviate this as $D_{\Lambda,\sigma}$ when $c = 0$. We note that $D_{\Lambda,\sigma}$ is $\sqrt{m}\sigma$-bounded.

Let $S^m$ denote the set of vectors in $\mathbb{R}^m$ whose length is 1. The norm of a matrix $R \in \mathbb{R}^{m \times m}$ is defined to be $\sup_{x \in S^m} ||Rx||$. The LWE problem was introduced by Regev [Reg05], who showed that solving it on average is as hard as (quantumly) solving several standard lattice problems in the worst case.

Definition 2.5 (LWE). For an integer $q = q(n) \geq 2$, and an error distribution $\chi = \chi(n)$ over $\mathbb{Z}_q$, the Learning With Errors problem $LWE_{n,m,q,\chi}$ is to distinguish between the following pairs of distributions (e.g. as given by a sampling oracle $O \in \{O_s, O^\dagger\}$):

\[
\{ A, s^T A + x^T \} \text{ and } \{ A, u \}
\]

where $A \overset{\$}{\leftarrow} \mathbb{Z}_q^{n \times m}$, $s \overset{\$}{\leftarrow} \mathbb{Z}_q^n$, $u \overset{\$}{\leftarrow} \mathbb{Z}_q^n$, and $x \leftarrow \chi^m$. 

16
**Gadget matrix.** The gadget matrix described below is proposed in [MP12, AP14].

**Definition 2.6.** Let \( m = n \cdot \lceil \log q \rceil \), and define the gadget matrix \( G = g_2 \otimes I_n \in \mathbb{Z}_q^{n \times m} \), where the vector \( g_2 = (1, 2, 4, \ldots, 2^{\lceil \log q \rceil}) \in \mathbb{Z}_q^{\lceil \log q \rceil} \). We will also refer to this gadget matrix as “powers-of-two” matrix. We define the inverse function \( G^{-1} : \mathbb{Z}_q^{n \times m} \to \{0, 1\}^{m \times m} \) which expands each entry \( a \in \mathbb{Z}_q \) of the input matrix into a column of size \( \lceil \log q \rceil \) consisting of the bits of binary representations. We have the property that for any matrix \( A \in \mathbb{Z}_q^{n \times m} \), it holds that \( G \cdot G^{-1}(A) = A \).

**Sampling Algorithms.** We will use the following algorithms to sample short vectors from specified lattices.

**Lemma 2.3 ([GPV08, AP10]).** Let \( q, n, m \) be positive integers with \( q \geq 2 \) and sufficiently large \( m = \Omega(n \log q) \). There exists a PPT algorithm \( \text{TrapGen}(q, n, m) \) that with overwhelming probability outputs a pair \( (A \in \mathbb{Z}_q^{n \times m}, T_A \in \mathbb{Z}_q^{m \times m}) \) such that the distribution of \( A \) is statistically close to uniform distribution over \( \mathbb{Z}_q^{n \times m} \) and \( T_A \) is a basis for \( \Lambda_q^1(A) \) satisfying
\[
\|T_A\| \leq O(n \log q) \quad \text{and} \quad \|\hat{T}_A\| \leq O(\sqrt{n \log q})
\]
extcept with \( \text{negl}(n) \) probability.

**Lemma 2.4 ([GPV08, CHKP10, ABB10]).** Let \( q > 2, m > n \). There are three sampling algorithms as follows:

- There is a PPT algorithm \( \text{SamplePre}(A, T_A, u, s) \), that takes as input: (1) a rank-\( n \) matrix \( A \in \mathbb{Z}_q^{n \times m} \), (2) a “short” basis \( T_A \) for lattice \( \Lambda_q^1(A) \), a vector \( u \in \mathbb{Z}_q^n \), (3) a Gaussian parameter \( s \) such that \( \|T_A\| \cdot \omega(\sqrt{\log(m)}) \); then outputs a vector \( r \in \mathbb{Z}^{m+1} \) distributed statistically close to \( D_{\Lambda_q^1(A),s} \).

- There is a PPT algorithm \( \text{SampleLeft}(A, B, T_A, u, s) \), that takes as input: (1) a rank-\( n \) matrix \( A \in \mathbb{Z}_q^{n \times m} \), and any matrix \( B \in \mathbb{Z}_q^{m \times m} \), (2) a “short” basis \( T_A \) for lattice \( \Lambda_q^1(A) \), a vector \( u \in \mathbb{Z}_q^n \), (3) a Gaussian parameter \( s \) such that \( \|T_A\| \cdot \omega(\sqrt{\log(m + m_1)}) \); then outputs a vector \( r \in \mathbb{Z}^{m+1} \) distributed statistically close to \( D_{\Lambda_q^1(A),s} \) where \( F := (A|B) \).

- There is a PPT algorithm \( \text{SampleRight}(A, B, R, T_B, u, s) \), that takes as input: (1) a matrix \( A \in \mathbb{Z}_q^{n \times m} \), and a rank-\( n \) matrix \( B \in \mathbb{Z}_q^{n \times m} \), a matrix \( R \in \mathbb{Z}_q^{m \times m} \), where \( s_R := \|R\| = \sup_{||x||=1} ||Rx|| \), (2) a “short” basis \( T_B \) for lattice \( \Lambda_q^1(B) \), a vector \( u \in \mathbb{Z}_q^n \), (3) a Gaussian parameter \( s \) such that \( \|T_B\| \cdot s_R \cdot \omega(\sqrt{\log m}) \); then outputs a vector \( r \in \mathbb{Z}^{2m} \) distributed statistically close to \( D_{\Lambda_q^1(F),s} \) where \( F := (A|AR + B) \).

Based on the above sampling algorithms, we have the following lemma:

**Lemma 2.5 ([GVW15c]).** Given integers \( n \geq 1, q \geq 2 \) there exists some \( m^* = m^*(n, q) = O(n \log q), \beta = \beta(n, q) = O(n \sqrt{\log q}) \) and \( s > \|T_A\| \cdot \omega(\sqrt{\log(m)}) \) such that for all \( m \geq m^* \) and all \( k \), we have
\[
A \approx A', \quad (A, T_A, U, V) \approx (A, T_A, U', V')
\]
where \( (A, T_A) \leftarrow \text{TrapGen}(q, n, m) \), \( A' \leftarrow \mathbb{Z}_q^{n \times m} \) and \( U \leftarrow D_{\mathbb{Z}^{m \times k}}, V = A \cdot U, V' \leftarrow \mathbb{Z}_q^{n \times k} \) and \( U' \leftarrow \text{SamplePre}(A, T_A, V', s) \).

17
2.4 Homomorphic Evaluation Algorithms

In this part, we recall three homomorphic evaluation algorithms \((\text{PubEval}, \text{TrapEval}, \text{CtEval})\). The following definition about homomorphic evaluation respective to some circuits is implicitly used in various constructions, such as attribute-based encryption [BGG+14, GV15] and predicate encryption [GVW15b].

**Definition 2.7** (\(\delta\)-expanding evaluation). The deterministic algorithms \((\text{PubEval}, \text{TrapEval}, \text{CtEval})\) are \(\delta\)-expanding with function (circuit with \(u\) inputs) \(f: \mathcal{X}^d \rightarrow \mathcal{Y}\) if they are efficient and satisfy the following properties:

- **PubEval**\((\{D_i \in \mathbb{Z}_{q}^{n \times m}\}_{i \in [d]}, f)\): On input matrices \(\{D_i\}_{i \in [d]}\) and a function \(f \in \mathcal{F}\), the public evaluation algorithm outputs \(D_f \in \mathbb{Z}_{q}^{n \times m}\) as the result.

- **TrapEval**\((x \in \mathcal{X}^d, A \in \mathbb{Z}_{q}^{n \times m}, \{R_i\}_{i \in [d]}, f)\): the trapdoor evaluation algorithm outputs \(R_f\), such that
  \[
  \text{PubEval}(\{AR_i + x_iG\}_{i \in [d]}, f) = AR_f + f(x)G
  \]

  Furthermore, we have \(||R_f|| \leq \delta \cdot \max_{i \in [d]} ||R_i||\).

- **CtEval**\((\{c_i\}_{i=1}^d, x, f)\): On input vectors \(\{c_i\}_{i=1}^d \in \mathbb{Z}_q^m\), an attribute \(x\) and function \(f\), the ciphertext evaluation algorithm outputs \(c_f(x) \in \mathbb{Z}_q^m\), such that

  \[
  \text{CtEval}(\{s^T(D_i + x_iG) + e_i\}_{i \in [d]}, x, f) = s^T(D_f + f(x)G) + e'
  \]

  where \(x = (x_1, ..., x_d)\) and \(D_f = \text{PubEval}(\{D_i \in \mathbb{Z}_{q}^{n \times m}\}_{i \in [d]}, f)\). Furthermore, we require \(||e'|| \leq \delta \cdot \max_{i \in [d]} ||e_i||\).

The definition can be extended to \(\delta\)-expanding with a family of functions \(\mathcal{F}\). I.e., \((\text{PubEval}, \text{TrapEval})\) are \(\delta\)-expanding with \(\mathcal{F}\) if and only if for all \(f \in \mathcal{F}\), the algorithms are \(\delta\)-expanding with \(f\).

3 Controlled Homomorphic Recoding Scheme

We propose a controlled homomorphic recoding scheme scheme consisting of probabilistic polynomial-time computable algorithms \(\text{CHR} = (\text{Setup}, \text{Enc}, \text{KeyEval}, \text{CtEval}, \text{ReEncKG}, \text{ReEnc}, \text{EqTest})\). Denote by \(\mathcal{S}\) to be the space of secret messages encrypted in the scheme. We first describe the basic algorithms.

- **Setup**, \(\text{CHR.Setup}(1^\lambda)\): On input security parameter \(\lambda\), it outputs a public key \(pk\) and secret key \(sk\).

- **Encoding procedure**, \(\text{CHR.Enc}(pk, y, s)\): On input public key \(pk\), public attribute \(y\) and secret message \(s\) from space \(\mathcal{S}\), it outputs the ciphertext \(ct\) (containing attribute \(y\)).

**Homomorphic Evaluation algorithms**: We describe the homomorphic evaluation algorithms below. The evaluation algorithm allows for homomorphically computing on the public keys and the attribute messages.

- **Homomorphic key evaluation**, \(\text{CHR.KeyEval}(\{pk_i\}_{i=1}^n, C)\): On input public keys \(\{pk_i\}_{i=1}^n\) and circuit \(C\), it homomorphically evaluates \(C\) with respect to \(\{pk_i\}_{i \in [n]}\) to obtain the resulting public key \(pk_C\).
• **Ciphertext evaluation**, CHR.CtEval(\(\{ct_i\}_{i=1}^n, \{yi\}_{i=1}^n, C\)): On input ciphertexts \(\{ct_i\}_{i\in[n]}\) under the public keys \(\{pk_i\}_{i\in[n]}\), circuit \(C\), it outputs the resulting ciphertext \(ct^*\).

Looking ahead, in the correctness definition, we require that all the ciphertexts \(ct_1, \ldots, ct_n\) are encoded using the same secret message \(s\).

**Controlled Recoding algorithms:** We describe the controlled recoding algorithms below. The recoding algorithm allows for translating ciphertexts which encode messages \(\{yi\}_{i\in[n]}\) generated using public keys \(\{pk_i\}_{i\in[n]}\) into a ciphertext of \(C(y_1, \ldots, y_n)\) under the target public key \(pk^*\). This translation is performed using a special recoding key \(rk\).

• **Recoding key generation**, CHR.ReEncKG \((pk_1, \ldots, pk_n, sk_i, pk^*, f)\): On input set of public keys \(\{pk_j\}_{j\in[n]}\), secret key \(sk_i\) for the \(i\)-th public key, target public key \(pk^*\), control function \(f\), it outputs the recoding key \(rk\).

• **Ciphertext recoding procedure**, CHR.ReEnc \((rk, \{(pk_i, ct_i)\}_{i\in[n]}\)\): On input recoding key \(rk\), ciphertexts \(ct_1, \ldots, ct_n\) computed under public keys \(pk_1, \ldots, pk_n\), it outputs the recoded ciphertext \(ct^*\).

**Auxiliary algorithm: Equality Test.** Finally, we describe an equality test algorithm. This determines if two ciphertexts corresponds to encryptions of the same attribute message and secret message.

• **Equality test**, CHR.EqTest \((pk, ct_1, ct_2)\): On input public key \(pk\), two ciphertexts \(ct_1, ct_2\), it outputs \(\text{Equal}\) if both \(ct_1\) and \(ct_2\) encrypt the same attribute using the same secret message and the same public key \(pk\). Otherwise, it outputs \(\text{NotEqual}\).

**Remark 3.1.** Looking ahead, in the construction of ABE for RAMs from CHR, we sample an anchor public key and secret key pair \((pk_0, sk_0)\) and this pair is used in the generation of all the recoding keys.

We describe the properties associated with a controlled homomorphic recoding scheme.

### 3.1 Correctness

We first begin with the correctness property. We describe some auxiliary algorithms that will be useful to describe the correctness and security properties.

**Derivation of Recoding Keys**, DerivReKey \((\{pk_i\}_{i\in[l]}, sk_{i*}, \{C_i\}_{i\in[L]}, pk^*, f)\): It takes as input public keys \(\{pk_i\}_{i\in[l]}\), secret key \(sk_{i*}\) for the \(i\)-th public key, target public key \(pk^*\) and control function \(f\), it does the following:

1. Evaluate public key, \(pk_{C_i} \leftarrow \text{CHR.KeyEval}(pk_1, \ldots, pk_{\ell}, C_i)\), for \(i \in [L]\).
2. Obtain \(rk\) by running \(\text{CHR.ReEncKG}(\{pk_{C_i}\}_{i\in[k]}, \{pk_{C_i}\}_{i\in[L]}, sk_{i*}, pk^*, f)\), where \(k \leq \ell\). That is, \(rk\) recodes ciphertexts encoded under the public keys \(\{pk_{C_i}\}_{i\in[k]}\) and \(\{pk_{C_i}\}_{i\in[L]}\).

**Derivation of Recoded Ciphertexts**, DerivReEnc \((rk, \{(pk_i, ct_i)\}_{i\in[l]}, \{C_i\}_{i\in[L]}\)\): It takes as input recoding key \(rk\), public keys \(\{pk_i\}_{i\in[l]}\), original ciphertexts \(\{ct_i\}_{i\in[l]}\), circuits \(\{C_i\}_{i\in[L]}\), it does the following:

---

*For ease of notation, we omit the public keys in the input to algorithm CHR.ReEnc when the context is clear.*
1. Evaluate public key, $pk_{C_i} \leftarrow \text{CHR.KeyEval}(pk_1, \ldots, pk_{\ell}, C_i)$, for $i \in [L]$.
2. Evaluate ciphertexts $ct_{C_i} \leftarrow \text{CHR.CtEval}(ct_1, \ldots, ct_{\ell}, C_i)$ for $i \in [L]$.
3. Compute the recoding algorithm, $\text{CHR.ReEnc}(rk, \{(pk_i, ct_i)\}_{i \in [k]}, \{(pk_{C_i}, ct_{C_i})\}_{i \in [L]})$, where $k \leq \ell$, to obtain the recoded ciphertext $ct^*$. Then output $ct^*$.

Put simply, the recoding key $rk$ recodes ciphertexts computed with respect to the public keys $\{pk_i\}_{i \in [k]}$ and $\{pk_{C_i}\}_{i \in [k]}$ into a ciphertext encoded with respect to the public key $pk^*$.

We explain the correctness property below. It incorporates the correctness of both the homomorphic and the recoding phases.

**Definition 3.1** (CHR Correctness). Consider a message $y \in \{0, 1\}^\ell$, secret message $s$ and ciphertexts $ct_1, \ldots, ct_\ell$, circuits $C_1, \ldots, C_L$ and a function $f$. Consider the following process:

1. Execute $\text{CHR.Setup}(1^\lambda)$, $\ell$ number of times to obtain $\ell$ public/secret key pairs $\{(pk_i, sk_i)\}_{i \in [\ell]}$. Also, compute target public key $\{(pk^*, sk^*)\} \leftarrow \text{CHR.Setup}(1^\lambda)$.
2. Compute $\text{DerivReKey}(\{(pk_i)_{i \in [\ell]}, sk_i, \{C_i\}_{i \in [L]}, pk^*, f\})$ to obtain the recoding key $rk$, for some $i^* \in [\ell]$.
3. Compute $\text{DerivReEnc}(rk, \{(pk_{i^*}, ct_{i^*})\}_{i \in [\ell]}, \{C_{i^*}\}_{i \in [L]})$ to obtain ciphertext $ct^*$.
4. Finally, compute the ciphertext $ct^*_{\text{fresh}} \leftarrow \text{CHR.Enc}(pk^*, f(C_1(y), \ldots, C_L(y)), s)$.

Suppose it holds that $\text{CHR.EqTest}(ct_i, \text{Enc}(pk_i, y_i, s)) = \text{Equal}$ with probability $1 - \text{negl}(\lambda)$, where $y_i$ is the $i$-th bit of $y$ and $s$ is uniformly random picked. Then it should hold that $\text{CHR.EqTest}(ct^*, ct^*_{\text{fresh}}) = \text{Equal}$ with probability $1 - \text{negl}(\lambda)$.

### 3.2 Security Definitions

The security definitions of controlled homomorphic recoding scheme $\Pi$ consists of four parts: indistinguishability of setup, indistinguishability of simulated keys, indistinguishability of recoding keys and pseudorandomness of ciphertexts. We describe them in detail below.

#### 3.2.1 Indistinguishability of Setup

This property intuitively states that the distribution of public keys produced by real setup is statistically close to that produced by simulated setup. We define the following simulated setup algorithm:

$\text{Sim.CHRSetup}(1^\lambda, z)$: It takes as input security parameter $\lambda$, input $z$ to be programmed and it outputs the programmed simulated public key $\text{Sim.pk}$ and associated trapdoor $\tau$.

**Definition 3.2** (Indistinguishability of Setup). A controlled homomorphic recoding scheme $\Pi$ is said to satisfy **indistinguishability of setup** if $\{pk\} \approx \{\text{Sim.pk}\}$ holds, where $(pk, sk) \leftarrow \text{CHR.Setup}(1^\lambda)$ and $(\text{Sim.pk}, \tau) \leftarrow \text{Sim.CHRSetup}(1^\lambda, z)$ for some $z \in \mathbb{Z}$.
3.2.2 Indistinguishability of Simulated Keys

We will first define a simulated key generation algorithm.

\[ \text{Sim.CHR}_{\text{key}}(pk, \{\text{Sim.pk}_i, \tau_i\}_{i \in [\ell]}, \{C_{ij}\}_{i \in [L], j \in [K]}, \{f_j\}_{j \in [K]}): \text{On input public keys } pk, \{\text{Sim.pk}_i\}_{i \in [\ell]} \]

\n
with associated trapdoors \( \{\tau_i\}_{i \in [\ell]} \), circuits \( \{C_{ij}\}_{i \in [L], j \in [K]} \), functions \( \{f_j\}_{j \in [K]} \), it outputs simulated recoding keys \( \{\text{Sim.rk}_i\}_{i \in [K]} \) and simulated target public keys \( \{\text{Sim.pk}_i^*\}_{i \in [K]} \).

We now define this property formally. This property states the output distribution of the following two procedures are statistically close:

- Simulating the original public keys (except one of the public keys \( pk \)). Generate the recoding keys and target public key honestly, with the help of \( sk \).
- Simulate the original public keys. Execute \( \text{Sim.CHR}_{\text{key}} \) to obtain the simulated recoding keys and the target public keys.

**Definition 3.3 (Indistinguishability of Simulated Keys).** A controlled homomorphic recoding scheme \( \Pi \) satisfies indistinguishability of simulated keys property with respect to a distribution \( \mathcal{E}_{\text{aux}} \) if the following holds: for any collection of circuits \( \{C_{ij}\}_{i \in [L], j \in [K]} \), control functions \( \{f_j\}_{j \in [K]} \),

\[
\{(pk_i^*, rk_j)_{j \in [K]}\} \approx \{(\text{Sim.pk}_j^*, \text{Sim.rk}_j)_{j \in [K]}\},
\]

where:

1. For \( j \in [K] \), compute the setup \( pk_i^* \leftarrow \text{CHR.Setup}(1^\lambda) \). Then compute normal setup algorithm \((pk, sk) \leftarrow \text{CHR.Setup}(1^\lambda)\). Next, for \( j \in [\ell] \), compute \((\text{Sim.pk}_j, \tau_j) \leftarrow \mathcal{E}_{\text{aux}}(1^\lambda, j)\).
2. For \( j \in [K] \), execute \( rk_j \leftarrow \text{DerivReKey}(pk, \{\text{Sim.pk}_i\}_{i \in [\ell]}, sk, \{C_{ij}\}_{i \in [L]}, pk_i^*, \{f_j\}_{j \in [K]}\) to obtain the simulated recoding keys \( \{\text{Sim.rk}_j\}_{j \in [K]} \) and simulated public keys \( \{\text{Sim.pk}_j^*\}_{j \in [K]} \) associated with trapdoors \( \{\tau_j^*\}_{j \in [K]} \).

We refer the reader to the technical overview for a brief explanation as to why \( \mathcal{E}_{\text{aux}} \) is necessary in the above definition.

3.2.3 Indistinguishability of Recoding Keys

This property intuitively says that it is hard to distinguish honestly generated recoding keys from simulated recoding keys. To define this formally, we first describe a simulated recoding key generation algorithm as follows:

\[ \text{Sim.CHR}_{\text{rk}}(pk, \{\text{Sim.pk}_i\}, \tau_i \in [\ell], \{C_i\}_{i \in [L]}, pk^*, P, f, aux): \text{On input public key } pk, \text{simulated public keys } \{\text{Sim.pk}_i\}_{i \in [\ell]} \text{ with associated trapdoors } \{\tau_i\}_{i \in [\ell]}, \text{circuits } \{C_i\}_{i \in [L]}, \text{target public key } pk^*, \text{predicate } P, \text{controlled function } f \text{ and auxiliary information } aux, \text{it outputs a simulated recoding key } rk_{\text{sim}} \text{ only if the output of } P(f, aux) = 1. \text{Otherwise, it outputs } \bot. \]

**Remark 3.2.** Looking ahead, in the construction of ABE for RAMs, \( P(f, aux) \) tests if the computation has terminated and if so, its output depends on the result of the computation. For instance, \( P(f, aux) \neq 1 \) if the computation has terminated and it outputs 0 (meaning that the message can be decrypted in this case). And thus, we should precisely be able to simulate in the scenario where the output of the computation is not 0 or if the computation has not terminated.
Definition 3.4 (Indistinguishability of recoding Keys). A controlled homomorphic recoding scheme \( \Pi \) satisfies indistinguishability of recoding keys property with respect to a distribution \( \mathcal{E}_{\text{aux}} \) and predicate \( P \), if \( \{ \text{rk}_{\text{sim}} \} \approx \{ \text{rk}_{\text{real}} \} \), where circuits \( C_1, \ldots, C_L \), function \( f \) such that \( P(f, \text{aux}) = 1 \), and we compute the distribution as:

- For \( i \in [\ell] \), compute the simulated setup \( (\text{Sim}.\text{pk}_i, \tau_i) \leftarrow \mathcal{E}_{\text{aux}}(1^\lambda, i) \). Compute the setup algorithm \( \text{CHR}.\text{Setup}(1^\lambda) \) to obtain the public key-secret key pairs \( (\text{pk}, \text{sk}) \) and \( (\text{pk}^*, \text{sk}^*) \).
- Compute \( \text{DerivReKey}(\text{pk}, \{ \text{Sim}.\text{pk}_i \}_{i \in [\ell]}, \text{sk}, \{ C_i \}_{i \in [L]}, \text{pk}^*, f) \) to obtain the recoding key \( \text{rk}_{\text{real}} \).
- Compute \( \text{Sim}.\text{CHR}_{\text{rk}}(\text{pk}, \{ \text{Sim}.\text{pk}_i, \tau_i \}_{i \in [\ell]}, \{ C_i \}_{i \in [L]}, \text{pk}^*, P, f, \text{aux}) \) to obtain the recoding key \( \text{rk}_{\text{sim}} \).

3.2.4 Pseudorandomness of Ciphertexts

Lastly, the pseudorandomness of ciphertexts requires that the distribution of ciphertexts is computationally close to the uniformly distribution over ciphertext space. We define the property formally below.

Definition 3.5 (Pseudorandomness of Ciphertexts). A controlled homomorphic recoding scheme \( \Pi \) is said to satisfy pseudorandomness of ciphertexts property if for any message \( y \in \mathbb{Z} \), it is computationally hard to distinguish \( \{ \text{Enc}(\text{pk}, y, s) \} \) from uniformly random distribution over ciphertext space, where \( (\text{pk}, \text{sk}) \leftarrow \text{Setup}(1^\lambda), s \leftarrow \mathcal{S} \).

4 Instantiation of CHR from Lattices

In this part, we show how to instantiate controlled homomorphic recoding scheme from lattices, particularly the LWE assumption (c.f. Definition 2.5). Then we prove the correctness of our instantiation and set the parameters in the following subsection. We describe the algorithms \( \text{CHR} = (\text{Setup}, \text{Enc}, \text{KeyEval}, \text{CtEval}, \text{ReEncKG}, \text{ReEnc}, \text{EqTest}) \) below: first, we start by describing the setup and the encoding algorithms. Then we describe the homomorphism phase, followed by equality test. We postpone the description of the recoding phase to the end. Our instantiation \( \Pi \) is as follows:

Basic algorithms: We describe setup and encoding algorithms below.

- \( \text{CHR}.\text{Setup}(1^\lambda) \): On input the security parameter \( \lambda \), the setup algorithm generates a matrix \( A \in \mathbb{Z}_q^{n \times m} \) along with its trapdoor \( T_A \) using

\[
(A, T_A) \leftarrow \text{TrapGen}(q, n, m)
\]

Output \( \text{pk} = A \) and \( \text{sk} = T_A \).

- \( \text{CHR}.\text{Enc}(\text{pk}, y, s) \): On input a public key \( \text{pk} = A \in \mathbb{Z}_q^{n \times m} \), an attribute bit \( y \in \{0, 1\} \) and a secret vector \( s \in \mathbb{Z}_q^n \), this encoding procedure outputs the ciphertext \( \text{ct} = (c = s^T(A + yG) + e^T, y) \), where \( e \leftarrow D_{\mathbb{Z}_q^{m \times \sigma}} \).
**Homomorphism Phase:** We describe the key evaluation and ciphertext homomorphism phase below.

- **CHR.KeyEval**(pk₁,..., pkₖ, C): On input \( \{ pk_i = A_i \}_{i=1}^k \) and a circuit \( C \), the algorithm outputs \( pk_C = \text{PubEval}(\{ A_i \}_{i \in [k]}, C) \).

- **CHR.CtEval**(ct₁,..., ctₖ, C): On input ciphertexts \( \{ ct_i = (c_i, y_i) \}_{i \in [k]} \) encrypting \( \{ y_i \}_{i \in [k]} \) under public key \( \{ pk_i \}_{i \in [k]} \) respectively and a circuit \( C \), the algorithm outputs \( ct = \text{CtEval}(\{ c_i \}_{i \in [k]}, \{ y_i \}_{i \in [k]}, C) \).

**Equality Test:** We describe the equality test below.

- **CHR.EqTest**(pk, ct₁, ct₂): On input \( pk = A \) and \( ct_1, ct_2 \) encrypting message under public key \( pk \), the algorithm outputs \( \text{Equal} \) if \( \text{Round}(ct_1 - ct_2) = 0 \), and \( \text{NotEqual} \) otherwise, where function \( \text{Round}(\cdot) \) is defined as

\[
\text{Round}(x) = \begin{cases} 
0, & \text{if } |x| < q/4 \\
1, & \text{otherwise}
\end{cases}
\]

The control functions we support for generating recoding keys are

\[
\{ f_{ij}(i', j', b) | i, i' \in [N], j, j' \in [T], b \in \{0, 1\} \}, \{ f_i(i', b) | i, i' \in [N] \}, \{ g_i(x) | i \in [N] \}, h(\cdot)
\]

with descriptions as

\[
f_{ij}(i', j', b) = \begin{cases} 
b, & \text{if } i = i' \land j = j' \\
\perp, & \text{otherwise}
\end{cases}, \quad f_i(i', b) = \begin{cases} 
b, & \text{if } i = i' \\
\perp, & \text{otherwise}
\end{cases} \quad (4)
\]

\[
g_i(x) = \begin{cases} 
i, & \text{if } C(x) = i - 1 \\
\perp, & \text{otherwise}
\end{cases}, \quad h(x) = \begin{cases} 
1, & \text{if } x = 0 \\
\perp, & \text{otherwise}
\end{cases} \quad (5)
\]

where \( C \) is a gadget circuit defined as \( C(x) = \sum_{i=1}^L x_i 2^i \), and \( x = (x_1, \ldots, x_L) \). Our recoding algorithm described below also support the identity function, \( \text{Ind}(x) = x \). We use notation \( \mathcal{F} \) to denote the set of control functions supported by our lattice-based instantiation, i.e.

\[
\mathcal{F} = \{ \text{Ind}, h, \{ g_i \}, \{ f_i \} \}
\]

**Remark 4.1.** Jumping ahead, the controlled functions defined above are used in different scenarios in the ABE scheme, particularly in the key generation algorithm. We use the following table to illustrate their relations.

**Recoding Phase.** We describe the recoding phase next. The details of algorithms (ReEncKG, ReEnc) are as follows. We abuse the notation of algorithm ReEncKG and ReEnc by allowing they taking into several different forms of input and executing differently with respect to the inputs.

- **CHR.ReEncKG** (Inp): On input Inp, consider the following two cases:
Table 2: Usage of Controlled Functions in ABE Setting

- **If Inp is of the form** \((pk_0, pk_1, sk_0, pk^*, \text{Ind})\): Here the controlled function \(\text{ind}\) denotes the identity function, i.e. \(\text{ind}(x) = x\) for any \(x\). The public keys \(pk_0, pk_1, pk^*\), secret key \(sk_0\) and target public key \(pk^*\) are parsed as follows,

\[
pk_0 = A_0, \quad pk_1 = A_1, \quad sk_0 = T_{A_0}, \quad pk^* = A^*
\]

the recoding key generation algorithm computes

\[
R \leftarrow \text{SamplePre}(A_0, T_{A_0}, A^* - A_1, \sigma)
\]

such that

\[
[A_0|A_1] \cdot [R_{I_m}] = A^*
\]

Then output \(rk = [R|I_m]\).

- **If Inp is of the form** \((pk_0, pk_1, sk_0, pk^*, h)\): The public keys \(pk_0, pk_1\), secret key \(sk_0\), target public key \(pk^*\), controlled function \(h\) (c.f. Equation (5)) are parsed as follows,

\[
pk_0 = A_0, \quad \{pk_i = A_i\}, \quad sk_0 = T_{A_0}, \quad pk^* = A^*
\]

The recoding key generation algorithm computes

\[
(R_0, R_1) \leftarrow \text{SampleLeft}(A_0, A_1, T_{A_0}, A^* + iG, \sigma)
\]

such that

\[
[A_0|A_1] \cdot \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = A^*
\]

Then output \(rk = (R_0, R_1)\).

- **if Inp is of the form** \((pk_0, \{pk_i, \}_{i=1}^L, sk_0, pk^*, gi)\): The public keys \(pk_0, \{pk_i\}_{i=1}^L\), secret key \(sk_0\), target public key \(pk^*\), controlled function \(gi\) (c.f. Equation (5)) are parsed as follows,

\[
pk_0 = A_0, \quad \{pk_i = A_i\}, \quad sk_0 = T_{A_0}, \quad pk^* = A^*
\]

the recoding key generation algorithm first computes \(A_C = \text{CHR.KeyEval}(pk_1, ..., pk_L, C)\), then computes

\[
(R_0, R_1) \leftarrow \text{SampleLeft}(A_0, A_C + (i - 1)G, T_{A_0}, A^* + iG, \sigma)
\]

such that

\[
[A_0|A_C + (i - 1)G] \cdot \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = A^* + iG
\]

Then output \(rk = (R_0, R_1)\).
- If \textbf{Inp} is of the form \((pk_0, pk_1, pk_2, sk_0, pk^*, f_i)\): The public keys \(pk_0, pk_1, pk_2\), secret key \(sk_0\), target public key \(pk^*\), function \(f_i\) (c.f. Equation (4)) are parsed as follows,

\[
\begin{align*}
    pk_0 &= A_0, & pk_1 &= A_1, & pk_2 &= A_2, & sk_0 &= T_{A_0}, & pk^* &= A^* \\
\end{align*}
\]

the recoding key generation algorithm first samples \(R_1 \leftarrow D_{\mathbb{Z}^{m \times m}, \sigma}\) and then computes

\[
R_0 \leftarrow \text{SampleLeft}(A_0, T_{A_0}, A^* - A_2 - (A_1 + iG)R_1, \sigma)
\]

such that

\[
[A_0|A_1 + iG|A_2] \cdot \begin{bmatrix} R_0 \\ R_1 \\ I_m \end{bmatrix} = A^*
\]

Then output \(rk_i = [R_0|R_1|I_m]\).

- If \textbf{Inp} is of the form \((pk_0, pk_1, pk_2, pk_3, sk_0, pk^*, f_{ij})\): The public keys \(pk_0, pk_1, pk_2, pk_3\), secret key \(sk_0\), target public key \(pk^*\), function \(f_{ij}\) (c.f. Equation (4)) are parsed as follows,

\[
\begin{align*}
    pk_0 &= A_0, & pk_1 &= A_1, & pk_2 &= A_2, & pk_3 &= A_3, & sk_0 &= T_{A_0}, & pk^* &= A^* \\
\end{align*}
\]

the recoding key generation algorithm first samples \(R_2 \leftarrow D_{\mathbb{Z}^{m \times m}, \sigma}\) and then computes

\[
[R_0|R_1] \leftarrow \text{SampleLeft}(A_0, A + iG, T_{A_0}, A^* - A_3 - (A_2 + jG)R_2, \sigma)
\]

such that

\[
[A_0|A_1 + iG|A_2 + jG|A_3] \cdot \begin{bmatrix} R_0 \\ R_1 \\ R_2 \\ I_m \end{bmatrix} = A^*
\]

Then output \(rk_{ij} = [R_0|R_1|R_2|I_m]\).

\textbf{CHR.ReEnc(\textbf{Inp})}: On input \textbf{Inp}, consider the following two cases:

- If \textbf{Inp} is of the form \((rk, pk_0, ct_0, pk_1, ct_1)\): The recoding key \(rk\), pairs \((pk_0, ct_0)\) and \((pk_1, ct_1)\) are parsed as follows,

\[
\begin{align*}
    rk &= [R|I_m], & (pk_0 = A_0, ct_0 = (c_0, 0)), & (pk_1 = A_1, ct_1 = (c_1, y_1))
\end{align*}
\]

the recoding algorithm computes

\[
c_2 = (c_0, c_1) \cdot [R]_{I_m}
\]

Output re-encrypted ciphertext \((c_2, y_1)\).

- If \textbf{Inp} is of the form \((rk, pk_0, ct_0, pk_1, ct_1)\): The recoding key \(rk\), pairs \((pk_0, ct_0)\) and \((pk_1, ct_1)\) are parsed as follows,

\[
\begin{align*}
    rk &= [R_0|R_1], & (pk_0 = A_0, ct_0 = (c_0, 0)), & (pk_1 = A_1, ct_1 = (c_1, 0))
\end{align*}
\]

the recoding algorithm computes

\[
c^* = (c_0, c_1) \cdot [R_0]_{R_1}
\]

Output re-encrypted ciphertext \((c^*, 0)\).
- If \textbf{Inp} is of the form \((rk, pk_0, ct_0, \{pk_j, ct_j\}_{j=1}^L)\): The recoding key \(rk\), pairs \((pk_0, ct_0)\) and \(\{pk_j, ct_j\}_{j=1}^L\) are parsed as follows,

\[
rk = [R_0|R_1], \ (pk_0 = A_0, ct_0 = (c_0, 0)), \ (pk_j = A_j, ct_j = (c_j, b_j))_{j=1}^L
\]

the recoding algorithm first computes \(c = CHR.CtEval(c_1, \ldots, ct_L, C)\), and then calculates

\[
c' = (c_0, c) \cdot \begin{bmatrix} R_0 \\ R_1 \\ I_m \end{bmatrix}
\]

For \(j \in [L]\), compute \(c'_j = CHR.CtEval(c', C_j)\), where circuit \(C_j'\) converts integer \(i\) to its \(j\)-th bit. Output re-encrypted message \(\{c'_j, b'_j\}_{j=1}^L\).

- If \textbf{Inp} is of the form \((rk, pk_0, ct_0, pk_1, ct_1, pk_2, ct_2)\): The recoding key \(rk\) and public key/ciphertext pairs \((pk_i, ct_i)\) for \(i \in \{0, 1, 2\}\) are parsed as follows,

\[
rk = [R_0|R_1|I_m], \ (pk_0 = A_0, ct_0 = (c_0, 0)), \ (pk_1 = A_1, ct_1 = (c_1, i)), \\
\quad \ (pk_2 = A_2, ct_2 = (c_2, y))
\]

the recoding algorithm computes

\[
c^* = (c_0, c_1, c_2) \cdot \begin{bmatrix} R_0 \\ R_1 \\ I_m \end{bmatrix}
\]

Output re-encrypted message \((c^*, y)\).

- If \textbf{Inp} is of the form \((rk, pk_0, ct_0, pk_1, ct_1, pk_2, ct_2, pk_3, ct_3)\): The recoding key \(rk\) and public key/ciphertext pairs \((pk_i, ct_i)\) for \(i \in \{0, 1, 2, 3\}\) are parsed as follows,

\[
rk = [R_0|R_1|R_2], \ (pk_0 = A_0, ct_0 = (c_0, 0)), \ (pk_1 = A_1, ct_1 = (c_1, i)), \\
\quad \ (pk_2 = A_2, ct_2 = (c_2, j)), \ (pk_3 = A_3, ct_3 = (c_3, y))
\]

the recoding algorithm computes

\[
c^* = (c_0, c_1, c_2, c_3) \cdot \begin{bmatrix} R_0 \\ R_1 \\ R_2 \\ I_m \end{bmatrix}
\]

Output re-encrypted message \((c^*, y)\).

### 4.1 Correctness and Parameters Setting

Next, we show the correctness of the instantiation \(\Pi\) from lattices.

**Lemma 4.2.** The above instantiation \(\Pi\) of CHR for supported controlled function \(F\) as defined in Equation (6) from lattices is correct (c.f. Definition 3.1) given the parameters setting below.

**Proof.** For \(i \in \{0\} \cup [\ell]\), let \(y_0 = 0, y = (y_1, \ldots, y_\ell) \in \{0, 1\}^\ell\) and \((pk_i = A_i, sk_i = T_{A_i}) \leftarrow CHR.Setup(1^\lambda)\). For \(i \in \{0\} \cup [\ell]\), encrypt the message as

\[
ct_i = (c_i, y_i) \leftarrow CHR.Enc(pk, y_i, s)
\]

where \(s \overset{\$}{\leftarrow} \mathbb{Z}_q^n\) and \(c_i = s^T(A_i + y_iG) + e_i^T\) and \(e_i \leftarrow \mathcal{D}_{\mathbb{Z}_{2n}}\). Then compute auxiliary algorithms \((DerivReKey, DerivReEnc)\) as
• **DerivReKey**(Inp): On input Inp, consider the following cases:

  - if Inp is of the form \( \{pk_i, \{pk_i\}_{i \in \mathbb{U}}, sk_0, C, pk^*, \text{Ind} \} \): first parse

    \[
    \{pk_i = A_i\}_{i \in \{0\} \cup \mathbb{U}}, \ sk_0 = T_{A_0}, \ pk^* = A^*
    \]

    the algorithm evaluates \( pk_C \leftarrow \text{CHR.KeyEval}(\{pk_i\}_{i \in \mathbb{U}}, C) \) and then computes \( rk \) by running \( \text{CHR.ReEncKG}(pk_0, pk_C, pk^*, sk_0, \text{Ind}) \).

  - if Inp is of the form \( \{pk_0, \{pk_i\}_{i \in \mathbb{U}}, sk_0, C, pk^*, h \} \): first parse

    \[
    \{pk_i = A_i\}_{i \in \{0\} \cup \mathbb{U}}, \ sk_0 = T_{A_0}, \ pk^* = A^*
    \]

    the algorithm evaluates \( pk_C \leftarrow \text{CHR.KeyEval}(\{pk_i\}_{i \in \mathbb{U}}, C) \) and then computes \( rk \) by running \( \text{CHR.ReEncKG}(pk_0, pk_C, pk^*, sk_0, h) \).

  - If Inp is of the form \( \{pk_0, \{pk_i\}_{i = 1}^{\mathbb{L}}, sk_0, pk^*, g_i\} \), then compute

    \[
    rk \leftarrow \text{CHR.ReEncKG}(pk_0, \{pk_i\}_{i = 1}^{\mathbb{L}}, sk_0, pk^*, g_i)
    \]

• **DerivReEnc**(Inp): On input Inp, consider the following cases:

  - If \( rk \leftarrow \text{DerivReKey}(pk_0, \{pk_i\}_{i \in \mathbb{U}}, sk_0, C, pk^*, \text{Ind}) \): First evaluate the public key/ciphertext

    \[
    pk_C = A_C = \text{CHR.KeyEval}(\{pk_i\}_{i \in \mathbb{U}}, C)
    \]

    \[
    c_C(y) = s^T(A_C + C(y)G) + e^T = \text{CHR.CtEval}(\{ct_i\}_{i = 1}^{\mathbb{L}}, C)
    \]

    Next, compute the recoding

    \[
    c^* = \text{CHR.ReEnc}(rk, pk_0, pk_C, ct_0, ct_C(y))
    \]

    \[
    = (s^T[A_0|A_C + C(y)G| + (e^T_0, e^T)] \begin{bmatrix} R \\ I_m \end{bmatrix})
    \]

    \[
    = s^T(A^* + C(y)G) + (e^T_0R + e^T)
    \]

    Output the recoded ciphertext \( ct^* = (c^*, C(y)) \).
- If \( rk \leftarrow \text{DerivReKey}(pk_0, \{pk_i\}_{i \in [\ell]}, sk_0, C, pk^*, h) \): First evaluate the public key/ciphertext

\[
pk_C = A_C = \text{CHR.KeyEval}(\{pk_i\}_{i \in [\ell]}, C)
\]

\[
c_{C^t(y)} = s^T(A_C + 0G) + e^T = \text{CHR.CtEval}(\{ct_i\}_{i=1}^\ell, C)
\]

Next, compute the recoding

\[
c^* = \text{CHR.ReEnc}(rk, pk_0, pk_C, ct_0, ct_{C^t(y)})
\]

\[
= (s^T[A_0|A_C + 0G] + (e_0^T, e^T)) \begin{bmatrix} R_0 \\ R_1 \end{bmatrix}
\]

\[
= s^T(A^* + 0G) + (e_0^T R_0 + e^T R_1)
\]

Output the recoded ciphertext \( ct^* = (c^*, 0) \).

- If \( rk \leftarrow \text{CHR.ReEncKG}(pk_0, \{pk_i\}_{i=1}^L, sk_0, pk^*, g_i) \): First evaluate the public key/ciphertext

\[
pk_C = A_C = \text{CHR.KeyEval}(\{pk_i\}_{i \in [L]}, C)
\]

\[
c_{C^x} = s^T(A_C + (i - 1)G) + e = \text{CHR.CtEval}(ct_1, ..., ct_L, C)
\]

Then, compute the recoding

\[
c^* = \text{CHR.ReEnc}(rk, pk_0, pk_C, ct_0, ct_{C^x})
\]

\[
= (s^T[A_0|A_C + (i - 1)G] + (e_0^T, e^T)) \begin{bmatrix} R_0 \\ R_1 \end{bmatrix}
\]

\[
= s^T(A^* + iG) + (e_0^T R_0 + e^T R_1)
\]

Next, for \( j \in [L] \), compute \( c_j^* = \text{CHR.CtEval}(c', C_j) \), where circuit \( C_j \) converts integer \( i \) to its \( j \)-th bit. Output recoding \( \{c_j^*, b_j\}_{j=1}^L \), where \( \{b_j\}_{j=1}^L \) is the bit-representation of \( i \).

- If \( rk \leftarrow \text{DerivReKey}(pk_0, pk_C, pk', pk^*, sk_0, f_j) \): First evaluate public key/ciphertext as

\[
pk_C = A_{C^t} \leftarrow \text{CHR.KeyEval}(\{pk_i\}_{i \in [\ell]}, C_i)
\]

\[
c_{C^t(y)} = s^T(A_C + C(y)G) + e^T \leftarrow \text{CtEval}(\{ct_i\}_{i=1}^\ell, C)
\]

where \( C(y) = i \). Let \( ct' = (c', b) = \text{Enc}(pk', b, s) \), and \( c' = s^T(A' + bG) + e^T \). Next, compute

\[
c^* = \text{CHR.ReEnc}(rk, pk_0, ct_0, \{pk_{C^t}, ct_{C^t(y)}\}_{i \in [2]}, pk', ct')
\]

\[
= (s^T[A_0|A_{C^t} + iG|A_C + jG|A' + bG] + (e_1^T, e'^T)) \begin{bmatrix} R_0 \\ R_1 \\ I_m \end{bmatrix}
\]

\[
= s^T(A^* + bG) + (e_1^T R_0 + e'^T R_1 + e'^T)
\]

Output recoded ciphertext as \( ct^* = (e^*, b) \).

- If \( rk \leftarrow \text{DerivReKey}(pk_0, \{pk_i\}_{i \in [\ell]}, pk', sk_0, C_i, C_2, pk^*, f_{ij}) \): For \( i \in [2] \), compute \( pk_{C_i} = A_{C_i} \leftarrow \text{CHR.KeyEval}(\{pk_i\}_{i \in [\ell]}, C_i) \). Then for \( i \in [2] \), evaluate ciphertexts as

\[
c_{C_i(y)} = s^T(A_C + C_i(y)G) + e_i^T \leftarrow \text{CtEval}(\{ct_i\}_{i=1}^\ell, C)
\]
Lemma 4.3. The instantiation \( \Pi \) of controlled homomorphic recoding scheme satisfies indistinguishability of setup (c.f. Definition 3.2).
Proof. The difference between normal setup algorithm Setup(1^\lambda) and simulated setup algorithm Sim.CHRSpecetup(1^\lambda, z) is that in Setup(1^\lambda), the pk = A \in \mathbb{Z}_q^{n \times m} is generated by algorithm TrapGen(q, n, m) and in Sim.CHRSpecetup(1^\lambda, z), we compute Sim.pk = \tilde{A} = A'S - zG, where matrices A' \in \mathbb{Z}_q^{n \times m}, S \in \{-1,1\}^{m \times m}. By property of algorithm TrapGen as stated in Lemma 2.3, the output distribution of A is statistically close to uniform distribution. By Leftover Hash Lemma 2.5, the distribution of A'S is statistically close to uniform distribution given the facts that A' \in \mathbb{Z}_q^{n \times m}, S \in \{-1,1\}^{m \times m}. Therefore, we have that the distribution \{pk\} is statistically close to \{Sim.pk\}. □

Generalization of Sim.GenCHRSetup: We can further generalize the simulated setup by augmenting its input as Sim.CHRSpecetup(1^\lambda, z, \ell; A), where z \in \{0,1\}^\ell, and A is used in a similar way as A' in algorithm Sim.CHRSpecetup(1^\lambda, z).

Sim.GenCHRSetup(Inp): On input Inp, consider the following two cases:

- If Inp is of form (1^\lambda, z), then run Sim.CHRSpecetup(1^\lambda, z).
- If Inp is of form (1^\lambda, z, \ell; A), then for i \in [\ell], choose S_i \in \{-1,1\}^{m \times m}, and set Sim.pk_i = A_i = AS_i - z_iG, where z_i is i-th bit of vector z. Output \{Sim.pk_i\}_{i \in [\ell]} and trapdoors \{S_i\}_{i=1}^\ell.

Similarly, we can also show that the distribution of public keys generalized by Sim.GenCHRSetup(Inp) is statistically close to the distribution of running normal setup algorithm \ell times.

Corollary 4.4. The instantiation of controlled homomorphic recoding scheme satisfies indistinguishability of setup.

The proof of the above corollary is very similar to the proof of Lemma 4.3, thus we omit it here.

Remark 4.5. Looking ahead, the sequence of integers z to be programmed in the generalized case corresponds to challenge database committed by the adversary in the ABE setting.

Corollary 4.6. The distribution of (regularly generated or simulated) public keys is indistinguishable from uniformly random distribution over space \mathbb{Z}_q^{n \times m}.

Proof. For regularly generated public keys, they are computed by algorithm TrapGen. By Lemma 2.5, the distribution of regularly generated public keys is statistically close to random distribution. For simulated public keys, they are computed as A'S - zG, where A' is chosen from random, and S is chosen from distribution \mathcal{D}_{\mathbb{Z}_q^{n \times m}}. Again by Lemma 2.5, the distribution of simulated keys is statistically close to random. □

4.2.2 Indistinguishability of Simulated Keys

We first describe the simulated key generation algorithm Sim.CHRSpecetup(pk, {Sim.pk_i, \tau_i}_{i \in [\ell]}, \{C_i\}_{i \in [L]}, \{\gamma_i\}_{i=1}^\ell, f).

The circuits \{C_i\}_{i \in [L]} are defined as C_i : \{0,1\}^\ell \rightarrow \{0,1\}. To be consistent with our instantiation of controlled homomorphic recoding scheme where we allow algorithm DerivReKey to take into two different forms of inputs (c.f. proof of Lemma 4.2). For simplicity, we only consider the case where L = 0, 1, 2 and variant controlled function f, which is how we define algorithm DerivReKey as above. This proof can be generalized to the case where L is any arbitrary integer.

Case. L = 0 and f = g_i (c.f. Equation (5)). Sim.CHRSpecetup(pk_i, {Sim.pk_i, \tau_i}_{i \in [\ell]}, \{\gamma_i\}_{i=1}^\ell, g_i): First parse part of the input as pk = A, Sim.pk_i = A_i = AS_i - b_iG, \tau_i = S_i, where C(\{\gamma_i\}_{i=1}^\ell) = i. Evaluate public key as A_C = AS_C = KeyEval(\{Sim.pk_i\}_{i \in [\ell]}, C)
Lemma 4.7. The indistinguishability of simulated keys property is proved below:

Case. \( L = 1 \) and \( f = \text{Ind} \). Sim.CHرك(pk, \{Sim.pk\}_i, \tau_i)_i, y, C, \text{Ind}]: First parse part of the input as \( pk = A, \text{Sim.pk}_i = A_i = A S_i - y_i G \) and \( \tau = S_i \). Evaluate public key as \( A_C = AS_C - C(y)G = \text{KeyEval}(\{\text{Sim.pk}\}_i, C) \), and sample a random matrix \( R \) from \( D_{2^{m \times m}} \). Output Sim.\( \text{rk} = [R | R_1] \), Sim.pk = \( A(R + S_C) - C(y)G \) and its trapdoor \( R + S_C \).

Case. \( L = 1 \) and \( f = f_i \) (c.f. Equation (4)). Sim.CHرك(key(pk, \{Sim.pk\}_i, \tau_i)_i, y, f, i): First parse part of the input as \( pk = A, \text{Sim.pk}_i = A_i = A S_i - y_i G, \tau = S_i \), and Sim.pk' = \( A' = A S' - b G \). Evaluate public key as \( A_C = AS_C - C(y)G = \text{KeyEval}(\{\text{Sim.pk}\}_i, C) \).

Case. \( L = 2 \) and \( f = f_{jk} \) (c.f. Equation (4)). Sim.CHرك(key(pk, \{Sim.pk\}_i, \tau_i)_i, y, b, C, \text{Ind}]: First parse part of the input as \( pk = A, \text{Sim.pk}_i = A_i = A S_i - y_i G, \) and Sim.pk' = \( A' = A S' - b G \). Evaluate public key as \( A_C = AS_C - C(y)G = \text{KeyEval}(\{\text{Sim.pk}\}_i, C) \), for \( i = 1, 2 \).

The indistinguishability of simulated keys property is proved below:

**Lemma 4.7.** The instantiation \( \Pi \) of controlled homomorphic recoding scheme satisfies indistinguishability of simulated keys (c.f. Definition 3.3) with respect to \( \mathcal{E}_{aux} \), where \( \mathcal{E}_{aux} \) is defined as follows:
• Case I: $E_{aux}$ is the same as Sim.CHRSetup$(1^λ, z)$, where aux corresponds to value being programmed, $z$.

• Case II: $E_{aux}$ is the same as Sim.CHRkey.

Proof. The difference between the normal key generation DerivReKey and simulated key generation Sim.CHRkey are summarized as below:

• In algorithm DerivReKey, the target public key $A^*$ is given as random matrix over $\mathbb{Z}_q^{n \times m}$ and recoding keys using the secret key $sk = T_A$ of $pk = A$ via algorithm SamplePre or SampleLeft.

• In algorithm Sim.CHRkey, the target public key is not given as input, but generated via variant computing methods as listed above, and recoding keys are sampled from Gaussian distribution $D_{\mathbb{Z}_q^{n \times m}}$ (via direct sampling or algorithm SampleRight).

By Leftover Hash Lemma 2.5, the distribution $(A, A^*)$ is statistically close to the distribution $(A, A_R0)$. And by the properties of algorithms SamplePre, SampleLeft and SampleRight as stated in Lemma 2.4, their outputs are statistically close to discrete Gaussian distribution $D_{\mathbb{Z}_q^{n \times m}}$. This statement holds simulated public keys from algorithm $E_{aux}$ (the two cases defined above), where the simulated public keys are computed from algorithms Sim.CHRkey or Sim.CHRSetup. Therefore, we have that $(\text{Sim.pk}^*, \text{rk}) \approx (\text{Sim.pk}, \text{Sim.rk})$, and thus we prove our instantiation of controlled homomorphic recoding scheme satisfies indistinguishability of simulated keys.

Generalization of Sim.CHRkey: We note that we can generalize algorithm Sim.CHRkey$(pk, \{\text{Sim.pk}_i\}_{i \in [\ell]}, pk', y, C_1, C_2, f_{jk})$ to generate a sequence of simulated recoding keys $\{\text{Sim.rk}_{ij}\}_{i \in [N], j \in [L]}$ (for some integers $N, L$ as the range of circuits $C_1, C_2$ respectively) and one target simulated key:

Sim.GenCHRkey$(pk, \{\text{Sim.pk}_i\}_{i \in [\ell]}, pk', y, C_1, C_2, N, L)$: First Evaluate $y_1 = C_1(y)$ and $y_2 = C(y)$. Then compute $(\text{Sim.pk}^*, \text{Sim.rk}_{y_1 y_2}) \leftarrow \text{Sim.CHRkey}(pk, \{\text{Sim.pk}_i\}_{i \in [\ell]}, pk', y, C_1, C_2, f_{y_1 y_2})$, where Sim.$pk^* = A^*$. Then for $j \in [L], k \in [N],$

• If $j = y_1$ and $k \neq y_2$, sample $R_{jk1}$ from $D_{\mathbb{Z}_q^{n \times m}}$, then compute

$$[R_{jk0} | R_{jk2}] \leftarrow \text{SampleRight}(A, G, SC_2, TG, A^* - ASC_1 R_{jk1}, \sigma)$$

• If $j \neq y_1$ and $k = y_2$, sample $R_{jk2}$ from $D_{\mathbb{Z}_q^{n \times m}}$, then compute

$$[R_{jk0} | R_{jk1}] \leftarrow \text{SampleRight}(A, G, SC_1, TG, A^* - ASC_2 R_{jk2}, \sigma)$$

• If $j \neq y_1$ and $k \neq y_2$, sample $R_{jk1}$ from $D_{\mathbb{Z}_q^{n \times m}}$, then compute

$$[R_{jk0} | R_{jk2}] \leftarrow \text{SampleRight}(A, G, SC_2, TG, A^* - (ASC_1 - (y_1 - j)G)R_{jk1}, \sigma)$$

Output simulated recoding keys and target public key as $\{\text{Sim.rk}_{ij}\}_{i \in [N], j \in [L]}$ and Sim.$pk$.

Corollary 4.8. The generalized instantiation of controlled homomorphic recoding scheme satisfies indistinguishability of simulated keys (c.f. Definition 3.3) with respect to $E_{aux}$ as defined in Lemma 4.7.

The proof of indistinguishability of simulated keys from the generalized algorithm is very similar to the proof of Lemma 4.7, thus we omit it here.

Remark 4.9. Looking ahead, the above generalization of the algorithm Sim.CHRkey will be used in the ABE scheme to generate the recoding keys for translating the output of the step circuit in the $i$-th step into the $(i + 1)$-th step.
4.2.3 Indistinguishability of Recoding Keys

We first describe the simulated recoding key generation algorithm $\text{Sim.CHRR}_{rk}(\text{pk}, \{\text{Sim.pk}_i\}_{i \in [\ell]}, \{C_i\}_{i \in [L]}, y, \text{pk}^*, f)$.

In the inputs, for $i \in [\ell]$, the circuit $C_i : \{0,1\}^{L} \rightarrow \{0,1\}$ and $f$ is defined as

$$ f : \{0,1\}^L \rightarrow \{0,1\}, \quad f(\{x_i\}_{i \in [L]}) = \begin{cases} 0, & \text{if } \sum_{i=1}^{L} \bar{x}_i = 1 \\ 1, & \text{otherwise} \end{cases} $$

(7)

Remark 4.10. Looking ahead, in the ABE scheme, the function $f$ will be used to signal whether the output of the computation is all zero. If the output is all zero (earlier than the upper time bound $T$), then $f$ outputs 0.

$\text{Sim.CHRR}_{rk}(\text{pk}, \{\text{Sim.pk}_i, \tau_i\}_{i \in [\ell]}, \{C_i\}_{i \in [L]}, y, \text{pk}^*, f)$: If $f(C_1(y), \ldots, C_L(y)) = 0$, then output $\bot$. Otherwise, parse the input as

$$ \text{pk} = A, \quad \{\text{Sim.pk}_i = A_i\}_{i \in [\ell]} \leftarrow \text{Sim.CHRSsetup}(1^\lambda, y, \ell; A), \quad \text{pk}^* = A^* $$

where $A_i = A S_i - y_i G$ and its trapdoor $\tau_i = S_i$. First for $i \in [L]$, evaluate public key as $A_{C_i} = A S_{C_i} - C_i(y) = \text{KeyEval}(\text{Sim.pk}_i, y, C_i)$. Since $f(\{C_i(y)\}_{i \in [L]}) = 1$, then there exists an index $k \in [L]$, such that $C_k(y) = 1$. For $i \in [L] - \{k\}$, sample matrices $R_i \leftarrow D_{\mathbb{Z}^{m \times m}, \sigma}$, and sample $[R_0| R_k]$, using

$$ [R_0| R_k] \leftarrow \text{SampleRight}(A, G, S_{C_k}, T_G, A^* - \sum_{i \in [L] - \{k\}} A_{C_i} R_i) $$

Output simulated recoding key as $\text{Sim.rk}_{sim} = \{R_i\}_{i \in [L]}$.

The indistinguishability of recoding keys property is proved using the properties of sampling algorithms used in the (simulated) recoding keys generation process as follows:

Lemma 4.11. The instantiation $\Pi$ of controlled homomorphic recoding scheme satisfies indistinguishability of recoding keys (c.f. Definition 3.4) with respect to $E_{aux}$, where $E_{aux}$ is defined in Lemma 4.7.

Proof. The only difference between generating normal recoding keys through $\text{DerivReKey}$ and simulated recoding keys through $\text{Sim.CHRR}_{rk}$ is

- The normal recoding keys are generated using algorithm $\text{SampleLeft}$ with $sk$.

- The simulated recoding keys are generated using algorithm $\text{SampleRight}$.

By the property of algorithms $\text{SampleLeft, SampleRight}$ as stated in Lemma 2.4, their outputs are statistically close to discrete Gaussian distribution. In simulated recoding generation, either the recoding keys are generated using $\text{SampleRight}$ or directly sampled from discrete Gaussian distribution. Therefore, we have $\{\text{rk}_{sim}\} \overset{c}{\approx} \{\text{rk}_{real}\}$, and thus show our instantiation of controlled homomorphic recoding scheme satisfies indistinguishability of recoding keys.

Generalization of $\text{Sim.CHRR}_{rk}$: We can generalized the above $\text{Sim.CHRR}_{rk}$ algorithm (used in the ABE construction), by evaluating public keys $\{A_{C_i}\}$ with respect to the gadget circuit $C$ before generating the simulated recoding keys. We define the algorithm formally as

$\text{Sim.GenCHR}_{rk}(\text{Inp})$: On input $\text{Inp}$, consider the following two cases:
• If Inp is of form \((pk, \{\text{Sim.pk}_i\}_{i \in [\ell]}, \{C_i\}_{i \in [L]}, y, pk^*, f)\), the run

\[\text{Sim.CH}_r(pk, \{\text{Sim.pk}_i\}_{i \in [\ell]}, \{C_i\}_{i \in [L]}, y, pk^*, f)\]

• If Inp is of form \((pk, \{\text{Sim.pk}_i\}_{i \in [\ell]}, \{C_i\}_{i \in [L]}, y, pk^*, f, c)\), then if \(f(C_1(y), \ldots, C_L(y)) = 0\), output \(\bot\). Otherwise, first for \(i \in [L]\), evaluate public key as \(A_{C_i} = AS_{C_i} - C_i(y)\) = \(\text{KeyEval}((\{\text{Sim.pk}_j\}_{j \in [\ell]}, C_i))\). Then evaluate \(\{A_{C_i}\}\) with respect to circuit \(C\) to obtain \(A_C = AS_C - zG\), where \(z \neq 0\) since \(f(C_1(y), \ldots, C_L(y)) = 1\). Sample \([R_0|R_1]\), using

\[\begin{align*}
[R_0|R_1] & \leftarrow \text{SampleRight}(A, G, S_C, T_G, A^*, \sigma) \\
\end{align*}\]

Output simulated recoding key \(\text{Sim.rk}_{\text{sim}} = [R_0|R_1]\).

Similarly, we can argue that the recoding key \(rk_{\text{sim}}\) key produced by the generalized algorithm \(\text{Sim.GenCHR}_r(\text{Inp})\) satisfies indistinguishability of recoding keys as

**Corollary 4.12.** The generalized instantiation of controlled homomorphic recoding scheme satisfies indistinguishability of recoding keys (c.f. Definition 3.4) with respect to \(E_{\text{aux}}\), where \(E_{\text{aux}}\) is defined in Lemma 4.7.

The proof of indistinguishability of simulated keys from the generalized algorithm is very similar to the proof of Lemma 4.11, thus we omit it here.

### 4.2.4 Pseudorandomness of Ciphertexts

We show pseudorandomness of ciphertexts (under simulated public keys or regularly generated public keys) based on the hardness of LWE assumption. We first describe the simulated encryption algorithm \(\text{Sim.CH}_r(\text{ct})\):

\(\text{Sim.CH}_r(\text{ct}, \text{Sim.pk}, \tau, y, s)\): On input the simulated public key \(\text{Sim.pk} = AS - yG\), the trapdoor \(S\), the attribute \(y\) and secret message \(s\), the simulated encryption algorithm computes and outputs the simulated ciphertext

\[\text{Sim.ct} = s^T(AS - yG + yG) + e^T S\]

where vector \(e \leftarrow D_{Z^{m, \sigma}}\).

**Lemma 4.13.** Assuming the hardness of sub-exponential LWE assumption (c.f. Definition 2.5), the instantiation \(\Pi\) of controlled homomorphic recoding scheme satisfies **pseudorandomness of ciphertexts** (c.f. Definition 3.5).

**Proof.** For regularly generated public keys, the ciphertext is of the form \(c = s^T(A - yG) + e^T\), where matrix \(A\) is chosen randomly from \(\mathbb{Z}^{n \times m}_q\), vector \(s\) is secret and chosen randomly from \(\mathbb{Z}^n_q\) and \(e \leftarrow D_{Z^{m, \sigma}}\). By hardness of LWE assumption, we have \(s^TA + e^T\) is computationally close to uniformly random distribution over \(\mathbb{Z}^m_q\). For simulated public keys, the ciphertext is of the form \(c = (s^T A + e^T)R\). By the LWE assumption, we have \(s^TA + e^T\) is computationally close to uniform distribution over \(\mathbb{Z}^m_q\), and since \(R \leftarrow D_{Z^{m \times m}}\) and by Leftover Hash Lemma, we have \((s^TA + e^T)R\) is computationally close to uniformly random distribution over \(\mathbb{Z}^m_q\).

Therefore, we prove that the instantiation satisfies the property of pseudorandomness of ciphertexts. 

\[\square\]
5 ABE for RAMs from CHR

In this section, we present the construction of ABE for the class of RAM programs $P$ from controlled homomorphic encoding scheme. Before we present our construction, we first define auxiliary circuits that will be associated with the step circuit of the RAM program.

**Auxiliary Circuits.** We define auxiliary circuits $(C_{up}, C_{ck})$ that will keep track of all the locations that have been written so far along with the most recent time step they were updated. These circuits will be useful to prevent an adversary from using an “illegal” encoding to recode to the next step. For instance, suppose the step circuit outputs the location 112 to be read in the next step. If the 112-th location has been written multiple times then the adversarial evaluator can use an ‘old’ encoding of the 112-th location (and hence, illegal) in the recoding step. We refer to this issue as repeated writing issue in the technical overview.

Thus, we have $(C_{up}, C_{ck})$ to keep track of the updates made. Moreover, the pair of circuits $(C_{up}, C_{ck})$ will be combined with the step circuit at the cost of increasing the size as a function of the upper bound $T$. We note that if there is not update happens in step $i$, we still need to add $(0, 0)$ to the update list to ensure the length grows with step number $i$.

We define auxiliary circuits $C_{up}$ and $C_{ck}$ as

| Input: a list $L$, location $i$, time $j$. |
| Computation: Traverse the list $L$ to check whether there is a pair $(i, j')$ where $j' < j$. If yes, replace the pair $(i, j')$ with $(i, j)$ and add $(0, 0)$ to the list. Otherwise, add $(i, j)$ to the list. |

**Figure 5: Definition of circuit $C_{up}$**

| Input: a list $L$, location $i$, time $j$. |
| Computation: Traverse the list $L$ to check whether there is a pair $(i, j)$. If yes, then return $j$, otherwise return 0. |

**Figure 6: Definition of circuit $C_{ck}$**

We assume that, for every program $P \in P$, the associated step circuit $C$ always takes as input the first location of memory, and the initial state is all 1. This is without loss of generality since every program $P$ can be modified such that this property holds, with the overhead of an extra time step. We list the RAM parameters that will be used in our construction in the following table:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>maximum database length</td>
<td>poly($\lambda$)</td>
</tr>
<tr>
<td>$T$</td>
<td>maximum running time</td>
<td>poly($\lambda$)</td>
</tr>
<tr>
<td>$\tau$</td>
<td>state bit-length</td>
<td>poly($\lambda$)</td>
</tr>
<tr>
<td>$\theta$</td>
<td>address bit-length</td>
<td>log $N$</td>
</tr>
<tr>
<td>$\eta$</td>
<td>update list unit bit-length</td>
<td>log $N + \log T$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>$C_{ck}$ circuit output bit-length</td>
<td>log $T$</td>
</tr>
</tbody>
</table>

**Table 3: ABE Parameters**
Every RAM program $P \in \mathcal{P}$ is parameterized by running time $t$ and memory length $N$ and represented as a step-circuit $C$, which is

$$(s_{t_i}, \text{loc}_{t_i}^w, b_{t_i}^r, \text{loc}_{t_i}^r) \leftarrow C(s_{t_{i-1}}, \text{loc}_{t_{i-1}}^r, b_{t_{i-1}}^r)$$

We incorporate the auxiliary circuits described above in the description of every program $P \in \mathcal{P}$. In more detail, we have a different step circuit for every step of the computation. The step-circuit $C_j$ in the $j$-th step, decomposed into binary representation can be written as follows, i.e.

$$C_j = \left( \{C_i^\tau \}_{i=1}^\tau, \{C_i^\theta \}_{i=1}^\theta, C_{\text{wb}}^\theta, \{C_i^\theta \}_{i=1}^\theta, \{C_i^{\text{up}} \}_{i=1}^{(j+1)\eta}, \{C_i^\phi \}_{i=1}^\phi \right)$$

where $C_i^\tau$ outputs the $i$-th bit of state for $i \in [\tau]$, $(C_i^\theta, C_i^\phi)$ outputs the $i$-th bit of the write/read address respectively for $i \in [\theta]$, and $C_{\text{wb}}^\theta$ outputs the bit to be written. Since the list maintained by update circuit $C_{\text{up}}$ increases by one component for each step, so for $j$-th step the number of decomposed outputs in $C_{\text{up}}$ is $(j+1)\eta$.

**Construction.** We construct attribute based encryption for RAMs from CHR scheme $\text{CHR} = (\text{Setup}, \text{Enc}, \text{KeyGen}, \text{CtEval}, \text{ReEncKG}, \text{ReEnc}, \text{EqTest})$. In our construction below, we define a gadget circuit $C$ as $C(x_1, ..., x_\theta) = \sum_{i=1}^\theta x_i 2^i$, where $x_i \in \{0, 1\}$ for $i \in [\theta]$. In the execution of RAM program, we assume that the initial state is all 1s, the satisfying state is all 0s, and the program always reads the 1st location of database. As mentioned in Table 2, we use different controlled functions in different ABE settings. We denote the ABE scheme to be $\text{ABE} = (\text{Setup}, \text{KeyGen}, \text{Enc}, \text{Dec})$.

**Notational convention:** As explained in the technical overview, we need a layer of public keys for every step of the computation. The 0-th layer of public keys and the target public key (for the last step of computation) is reused across different encryptions. The intermediate layers of public keys, however, are freshly sampled from one execution of key generation to another.

We use the superscript in the notation of public keys to denote the type of the value being encoded. The subscript denotes the index of the binary representation of the value being encoded.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step$[j]$.pk$^\tau$</td>
<td>$i$-th bit of state in $j$-th step</td>
</tr>
<tr>
<td>Step$[j]$.pk$^r$</td>
<td>$i$-th bit of to-read address for $(j+1)$-th step</td>
</tr>
<tr>
<td>Step$[j]$.pk$^u$</td>
<td>$i$-th bit of update list until $j$-th step</td>
</tr>
<tr>
<td>Step$[j]$.pk$^b$</td>
<td>$i$-th bit of database in $j$-th step</td>
</tr>
<tr>
<td>Step$[j]$.pk</td>
<td>reading value in $j$-th step</td>
</tr>
<tr>
<td>pk$^t_{\text{out}}$</td>
<td>target public key</td>
</tr>
</tbody>
</table>

Table 4: ABE Construction Notations

We use the notation $\text{Step}[j].\text{hompk}$ to correspond to the public key obtained by homomorphically evaluating on $j$-th layer public keys. The superscripts and subscripts on $\text{hompk}$ hold the same meaning as above.

- **$\text{ABE.Setup}(1^\lambda, 1^T)$**: On input security parameter $\lambda$ and time bound $T$,
  - **Public Keys associated with State**: Generate the 0-th step public keys that are used to encode the initial state. Compute $\text{CHR.Setup}(1^\lambda)$, $\tau$ number of times, to obtain $\{(\text{Step}[0].pk^\tau_i, \text{Step}[0].sk^\tau_i)\}_{i \in [\tau]}$. 

36
Public Keys associated with Read Address: Generate the 0-th step public keys that is used to encode the initial read address. Compute CHR.Setup(1^λ), \( \theta \) number of times, to obtain \( \{(\text{Step}[0], \text{pk}_{j_{\text{ra}}}[0], \text{Step}[0], \text{sk}_{i_{\text{ra}}}[0])\}_{j \in \{\theta\}} \).

Public Keys associated with Address List: Generate the 0-th step public keys that are used to encode the address list, which is initialized with zeroes. During the evaluation process, the address list is populated with the addresses written so far and the most recent time step they were written. Compute CHR.Setup(1^λ), \( \eta \) number of times, to obtain \( \{(\text{Step}[0], \text{pk}_{k, i_{\text{ra}}}[0], \text{Step}[0], \text{sk}_{k, i_{\text{ra}}}[0])\}_{k \in \{\eta\}} \). Generate \( \text{Step}[0] \cdot \text{pk}^t \) to encode the 0-th time step.

Public Keys associated with Database: Generate the 0-th step public keys that is used to encode the initial attribute database. Compute CHR.Setup(1^λ), \( N \) number of times, to obtain \( \{(\text{Step}[0], \text{pk}_{i_{\text{db}}}[0], \text{Step}[0], \text{sk}_{i_{\text{db}}}[0])\}_{i \in \{N\}} \).

Anchor Public Key: Generate a public key-secret key pair \( (\text{pk}_0, \text{sk}_0) \leftarrow \text{CHR.Setup}(1^\lambda) \). The public key \( \text{pk}_0 \) will participate in every recoding key process (during key generation), in which the secret key \( \text{sk}_0 \) will be used. We note that the secret keys generated in the above bullets will be discarded and only the public keys will be used for the rest of the construction.

Output master secret key \( \text{msk} = \text{sk}_0 \) and public parameter \( \text{pp} \) as
\[
\text{pp} = (\{(\text{Step}[0], \text{pk}_{st}[0])_{i \in \{\tau\}}, (\text{Step}[0], \text{pk}_{ra}[0])_{j \in \{\theta\}}, (\text{Step}[0], \text{pk}_{lt}[0])_{k \in \{\eta\}}, (\text{Step}[0], \text{pk}_{db}[0])_{i \in \{N\}}, \text{Step}[0], \text{pk}^t, \text{pk}_0, \text{pk}_{\text{out}}\})
\]

- \text{ABE.KeyGen}(\text{msk}, P): On input master secret key \( \text{msk} \) and program \( P \) with upper time bound \( T \) and database length \( N \), the key generation algorithm parse the step circuit of program \( P \) as \( (\{C_{st}^i\}_{j=1}^T \cup \{C_{ra}^i\}_{j=1}^T \cup \{C_{db}^i\}_{j=1}^T) \). Then generate public keys along for each step as:

- Public Keys associated with Read Value: For \( i \in \{T\} \), generate the public key associated with read value for each step. Compute CHR.Setup(1^λ), \( T \) number of times, to obtain \( \{(\text{Step}[i], \text{pk}, \text{Step}[i], \text{sk})\}_{i \in \{T\}} \).

- Public Keys associated with Time Step: For \( i \in \{T\} \), generate the public key associated with time step for each step. Compute CHR.Setup(1^λ), \( T \) number of times, to obtain \( \{(\text{Step}[i], \text{pk}^t, \text{Step}[i], \text{sk}^t)\}_{i \in \{T\}} \).

- Public Keys associated with State: For \( i \in \{T\} \), generate the i-th public keys that are used to encode the state. Compute CHR.Setup(1^λ), \( T \tau \) number of times, to obtain \( \{(\text{Step}[i], \text{pk}_{st}[i], \text{Step}[i], \text{sk}_{st}[i])\}_{i \in \{T\}, j \in \{\tau\}} \).

- Public Keys associated with Read Address: For \( i \in \{T\} \), generate the i-th public keys that are used to encode the read address for each step. Compute CHR.Setup(1^λ), \( T \theta \) number of times, to obtain \( \{(\text{Step}[i], \text{pk}_{ra}[i], \text{Step}[i], \text{sk}_{ra}[i])\}_{i \in \{T\}, j \in \{\theta\}} \).

- Public Keys associated with Database: For \( i \in \{T\} \), generate the i-th public keys that are used to encode the database for each step. Compute CHR.Setup(1^λ), \( T \eta \) number of times, to obtain \( \{(\text{Step}[i], \text{pk}_{db}[i], \text{Step}[i], \text{sk}_{db}[i])\}_{i \in \{T\}, \ell \in \{\eta\}} \).

- Public Keys associated with Address List: For \( i \in \{T\} \), generate the public keys that are used to encode the updated address list. In each step, the list grows by \( \eta \) entries as specified in the Definition of circuit \( C^\text{up} \) (c.f. Figure 5). For \( i \in \{T\} \), compute CHR.Setup(1^λ), \( (i + 1) \eta \) number of times, to obtain \( \{(\text{Step}[i], \text{pk}_{lt}[i], \text{Step}[i], \text{sk}_{lt}[i])_{j \in \{i + 1\}, \ell \in \{\eta\}} \).

For \( i \in \{T\} \), generate the recoding key for time-step as
\[
\text{Step}[i] \cdot \text{rk}^t \leftarrow \text{CHR.ReEncKG}(\text{pk}_0, \text{Step}[i], \text{pk}^t, \text{sk}_0, \text{Step}[i + 1], \text{pk}^t, g_i)
\]
where control function \( g_i \) is defined in Equation (5). Next, for \( i \in \{0\} \cup \{T\} \), do the following:
1. **STATE CIRCUIT** \( \{C^j_i\}_{j \in \{\tau\}} \): First for \( j \in \{\tau\} \), homomorphically compute public key \( \text{Step}[i].\text{hompk}^{j_\text{st}} \) with respect to \( C^j_j \), then evaluate the gadget circuit \( C \) on input the homomorphic public keys, and provide a terminating recoding key \( \text{Step}[i].\text{rk}^{\text{out}}_j \) from current \( i \)-th step to the output step. Next, provide a recoding key \( \text{Step}[i].\text{rk}^{\text{st}}_j \) which recode the state information of \( i \)-th step to the \((i+1)\)-th step. The detail follows: For \( j \in \{\tau\} \), evaluate \( C^j_j \)

\[
\text{Step}[i].\text{hompk}^{j_\text{st}} \leftarrow \text{CHR.KeyEval}(\{\text{Step}[i].\text{pk}^{k_\text{st}}_k\}_{k \in \{\tau\}}, \{\text{Step}[i].\text{pk}^{k_\text{st}}_k\}_{k \in \{\theta\}}, \text{Step}[i].\text{pk}, C^j_j)
\]

And then compute \( \text{Step}[i].\text{hompk}^{j_\text{st}} = \text{CHR.KeyEval}(\{\text{Step}[i].\text{hompk}^{j_\text{st}}_j\}_{j \in \{\tau\}}, C) \), and use the secret key \( \text{sk}_0 \) to compute the recoding key

\[
\text{Step}[i].\text{rk}^{\text{out}}_j \leftarrow \text{CHR.ReEncKG}(\text{pk}_0, \text{Step}[i].\text{hompk}^{j_\text{st}}, \text{sk}_0, \text{pk}^{\text{out}}_0, h)
\]

where control function \( h \) is defined in Equation (5). Next generate the recoding key as

\[
\text{Step}[i].\text{rk}^{\text{st}}_j \leftarrow \text{CHR.ReEncKG}(\text{pk}_0, \text{Step}[i].\text{hompk}^{j_\text{st}}, \text{sk}_0, \text{Step}[i+1].\text{pk}^{j_\text{st}}, \text{Ind})
\]

2. **READING ADDRESS CIRCUIT** \( \{C^j_r\}_{j \in \{\theta\}} \): First for \( j \in \{\theta\} \), homomorphically compute public key \( \text{Step}[i].\text{hompk}^{j_\text{ra}}_j \) with respect to \( C^j_j \), then provide a recoding key \( \text{Step}[i].\text{rk}^{j_\text{ra}}_j \) which recode the read address information of \( i \)-th step to the \((i+1)\)-th step. Next, evaluate the gadget circuit \( C \) on input the homomorphic public keys \( \{\text{Step}[i].\text{hompk}^{j_\text{ra}}_j\} \) to obtain \( \text{Step}[i].\text{pk}^{j_\text{ra}} \), and evaluate the check circuit \( C^{j_\text{ck}} \) on address list \( \{\text{Step}[i].\text{pk}^{j_\text{lk}}_k\}_{k \in \{\theta\}} \) and \( \text{Step}[i].\text{pk}^{j_\text{ra}} \). Provide recoding keys, which recode the specific database location to read value, according to read address \( \text{Step}[i].\text{pk}^{j_\text{ra}} \) and result \( \text{Step}[i].\text{pk}^{j_\text{ck}} \) of \( C^{j_\text{ck}} \). The detail follows: For \( j \in \{\theta\} \), evaluate \( C^j_j \) as

\[
\text{Step}[i].\text{hompk}^{j_\text{ra}}_j \leftarrow \text{CHR.KeyEval}(\{\text{Step}[i].\text{pk}^{j_\text{st}}_k\}_{k \in \{\tau\}}, \{\text{Step}[i].\text{pk}^{j_\text{st}}_k\}_{k \in \{\theta\}}, \text{Step}[i].\text{pk}, C^j_j)
\]

Then compute the following

\[
\text{Step}[i].\text{rk}^{j_\text{ra}}_j \leftarrow \text{CHR.ReEncKG}(\text{pk}_0, \text{Step}[i].\text{hompk}^{j_\text{ra}}_j, \text{sk}_0, \text{Step}[i+1].\text{pk}^{j_\text{ra}}, \text{Ind})
\]

Then evaluate gadget circuit \( C \) and the check circuit \( C^{j_\text{ck}} \) (c.f. Figure 6) as

\[
\text{Step}[i].\text{pk}^{j_\text{ra}} = \text{CHR.KeyEval}(\{\text{Step}[i].\text{hompk}^{j_\text{ra}}_j\}_{j \in \{\theta\}}, C)
\]

\[
\text{Step}[i].\text{pk}^{j_\text{ck}} \leftarrow \text{CHR.KeyEval}(\text{Step}[i].\text{pk}^{j_\text{ra}}, \{\text{Step}[i].\text{pk}^{j_\text{lk}}_k\}_{k \in \{\theta\}}, \text{Step}[i].\text{pk}^{j_\text{ck}}, C^{j_\text{ck}})
\]

Next, for \( k \in \{N\}, \ell \in \{i-1\} \), compute the following

\[
\text{Step}[i].\text{rk}^{j_\text{ra}}_{k_\ell} \leftarrow \text{CHR.ReEncKG}(\text{pk}_0, \text{Step}[i].\text{pk}^{j_\text{ra}}, \text{Step}[i].\text{pk}^{j_\text{ck}}, \text{Step}[\ell].\text{pk}^{j_\text{cb}}, \text{sk}_0, \text{Step}[i+1].\text{pk}, f_{k_\ell})
\]

where \( \{\text{Step}[\ell].\text{pk}^{j_\text{cb}}_{k_\ell}\}_{k \in \{N\}, \ell \in \{i-1\}} \) are freshly generated public keys in Writing address/value part as described below, and control function \( f_{k_\ell} \) is defined in Equation (4).

3. **WRITING ADDRESS/VALUE CIRCUITS** \( \{C^w_j\}_{j \in \{\theta\}}, C^{\text{wb}} \): First for \( j \in \{\theta\} \), homomorphically compute public key \( \text{Step}[i].\text{hompk}^{j_\text{w}}_j \) with respect to \( C^w_j \) and \( \text{Step}[i].\text{pk}^{\text{wb}}_j \) with respect to \( C^{\text{wb}} \). Then, evaluate the gadget circuit \( C \) on input the homomorphic public keys \( \{\text{Step}[i].\text{hompk}^{j_\text{w}}_j\} \) to obtain \( \text{Step}[i].\text{pk}^{j_\text{w}} \). Next, for each entry of the database, provide a recoding key, which recodes the writing value to the freshly generated database public key \( \text{Step}[i+1].\text{pk}^{j_\text{cb}}_j \). The detail follows: For \( j \in \{\theta\} \), evaluate \( C^w_j \) and \( C^{\text{wb}} \) as

\[
\text{Step}[i].\text{hompk}^{j_\text{w}}_j \leftarrow \text{CHR.KeyEval}(\{\text{Step}[i].\text{pk}^{j_\text{st}}_k\}_{k \in \{\tau\}}, \{\text{Step}[i].\text{pk}^{j_\text{st}}_k\}_{k \in \{\theta\}}, \text{Step}[i].\text{pk}, C^w_j)
\]
Step[i].pk^{wb} \leftarrow \text{CHR.KeyEval}([\text{Step}[i].pk^{sl}_{k[\tau]}]_{k \in [\tau]}, [\text{Step}[i].pk^{ra}_{k[\theta]}]_{k \in [\theta]}, \text{Step}[i].pk, C^{wb})

Then compute \text{Step}[i].pk^{w} = \text{CHR.KeyEval}([\text{Step}[i].\text{hompk}_{j}]_{j \in [\theta]}, C)$, where circuit $C$ is the gadget circuit. Then do following computation for $\ell \in [N]$

\text{Step}[i].rk_{\ell}^{w} \leftarrow \text{CHR.ReEncKG}(pk_{0}, \text{Step}[i].pk^{w}, \text{Step}[i].pk^{wb}, sk_{0}, \text{Step}[i + 1].pk^{db}, f_{\ell})$

where control function $f_{\ell}$ is defined in Equation (4).

4. UPDATE CIRCUIT $\{C^{up}_{j}\}_{j \in [(i+1)\eta]}$: First for $j \in [(i+1)\eta]$, homomorphically compute public key \text{Step}[i].\text{hompk}_{j}^{\ell}$ with respect to $C^{up}_{j}$. Then, provide a recoding key \text{Step}[i].rk_{j}^{\ell}$ which recode the address list information of $i$-th step to the $(i+1)$-th step. The detail follows: For $j \in [(i+1)\eta]$, evaluate the update circuit $C^{up}_{j}$ (c.f. Figure 5) as

\text{Step}[i].\text{hompk}_{j}^{\ell} \leftarrow \text{CHR.KeyEval}([\text{Step}[i].pk^{sl}_{j}]_{j \in [\eta]}, \text{Step}[i].pk^{w}, \text{Step}[i].pk^{l}, C^{up}_{j})$

Then use the secret key $sk_{0}$ to generate recoding key as

\text{Step}[i].rk_{j}^{\ell} \leftarrow \text{CHR.ReEncKG}(pk_{0}, \text{Step}[i].\text{hompk}_{j}^{\ell}, sk_{0}, \text{Step}[i + 1].pk^{l}_{j}, \text{Ind})

Output the secret key for RAM program $P$ as $sk_{P} = (P, \{\text{Step}[i].\text{KEY}\}_{i = 0}^{T-1},)$ where

\text{Step}[i].\text{KEY} = (\text{Step}[i].rk_{\text{out}}, \text{Step}[i].rk^{l}, [\text{Step}[i].rk^{st}_{j}]_{j \in [\tau]}, [\text{Step}[i].rk^{l}_{j}]_{j \in [\theta]},
\{\text{Step}[i].rk^{la}_{kj}]_{k \in [i-1], j \in [N]}, \{\text{Step}[i].rk^{w}_{j}]_{j \in [\eta]}, \{\text{Step}[i].rk_{j}^{\ell}\}_{j \in [(i+1)\eta]})

- ABE.Enc(pp, D, $\mu$): On input the public parameter pp, a database $D = \{x_{i}\}_{i=1}^{N}$ and a message $\mu$, the encryption algorithm first picks a secret message $s$ uniformly at random from $S$ and compute the following ciphertexts:

- DATABASE ENCRYPTION: For $i \in [N]$, generate ciphertexts for each entry of the database. Compute $\text{Step}[0].ct^{db}_{i} = \text{CHR.Enc}(\text{Step}[0].pk^{db}, x_{i}, s)$, for $i \in [N]$.

- INITIAL STATE ENCRYPTION: For $i \in [\tau]$, generate ciphertexts for initial state. Compute $\text{Step}[0].ct^{s}_{i} = \text{CHR.Enc}(\text{Step}[0].pk^{s}_{i}, 1, s)$, for $i \in [\tau]$.

- INITIAL READING ADDRESS ENCRYPTION: For $i \in [\theta]$, generate ciphertexts for initial reading address. Compute $\text{Step}[0].ct^{\theta}_{i} = \text{CHR Enc}(\text{Step}[0].pk^{\theta}_{i}, 1, s)$, and for $i = 2, \ldots, \theta$ compute $\text{Step}[0].ct^{\theta}_{i} = \text{CHR.Enc}(\text{Step}[0].pk^{\theta}_{i}, 0, s)$.

- INITIAL ADDRESS LIST ENCRYPTION: For $i \in [\eta]$, generate ciphertexts for initial address list. Compute $\text{Step}[0].ct^{\eta}_{i} = \text{CHR.Enc}(\text{Step}[0].pk^{\eta}_{i}, 0, s)$, for $i \in [\eta]$.

- AUXILIARY INFORMATION ENCRYPTION: Encrypt under the anchor public key $c_{0} = \text{CHR.Enc}(pk_{0}, 0, s)$ and 0-th time step public key $u_{0} = \text{CHR.Enc}(\text{Step}[0].pk^{l}, 0, s)$.

- MESSAGE ENCRYPTION: If the message $\mu = 0$, then compute $\psi = \text{Enc}(pk^{\text{out}}_{0}, 0, s)$, Otherwise, if the message $\mu = 1$, choose a random vector over the ciphertext space of CHR.

Output the ciphertext as

$ct_{D} = (D, c_{0}, u_{1}, [\text{Step}[0].ct^{sl}_{i}]_{i=1}^{\tau}, [\text{Step}[0].ct^{db}_{i}]_{i=1}^{N}, [\text{Step}[0].ct^{ra}_{i}]_{i=1}^{\theta}, [\text{Step}[0].ct^{l}_{i}]_{i=1}^{\eta}, \psi)$

- ABE.Dec(sk$_{P}$, ct$_{D}$): On input secret key sk$_{P}$ for RAM program $P$ and a ciphertext ct$_{D}$, outputs $\perp$ if $P^{D} \neq 0$. Otherwise, parse sk$_{P} = (P, \{\text{Step}[i].\text{KEY}\}_{i = 0}^{T-1})$. Parse the step circuit $C_{i}$ of the RAM program as

$C_{i} = ([C^{sl}_{j}]_{j=1}^{\tau}, [C^{w}_{i}]_{j=1}^{\theta}, [C^{wb}_{i}]_{j=1}^{\theta}, [C^{ra}_{j}]_{j=1}^{\theta}, [C^{up}_{j}]_{j \in [(i+1)\eta]})$
1. **State Circuit** \(\{C_j^i\}_{j \in [\tau]}\): First, for \(j \in [\tau]\), homomorphically compute ciphertext \(\text{Step}[i].\text{homct}_j^i\) with respect to \(C_j^i\). If at the current step, the state is all 0, the use the terminating key \(\text{Step}[i].\text{rk}^{\text{out}}\) to recode the \(i\)-th step to the output step, and execute the last step algorithm. Otherwise, use the recoding key \(\text{Step}[i].\text{rk}^{\text{st}}\) to recode the state information of \(i\)-th step to \((i+1)\)-th step. The detail follows: For \(j \in [\tau]\), evaluate \(C_j^i\) as

\[
\text{Step}[i] . \text{homct}_j^i \leftarrow \text{CHR.CtEval}((\{\text{Step}[i].ct_k^i\}_{k \in [\tau]}, \{\text{Step}[i].ct_{ra}^i\}_{k \in [\theta]}, \text{Step}[i].ct, C_j^i))
\]

If at the current step, the state is all 0, then evaluate the following function \(\text{Step}[i].ct_{st} = \text{CHR.CtEval}((\{\text{Step}[i].\text{homct}_j^i\}_{j \in [\tau]}, \text{C}))\), compute

\[
\text{Step}[i].ct_{\text{out}} = \text{CHR.ReEnc}(c_0, \text{Step}[i].ct_{st}, \text{Step}[i].\text{rk}_{\text{out}}^i) \tag{7}
\]

and, jump to the last step. Otherwise, for \(j \in [\tau]\), compute

\[
\text{Step}[i+1].ct_{st} = \text{CHR.ReEnc}(c_0, \text{Step}[i].\text{homct}_j^i, \text{Step}[i].\text{rk}_{st}^i)
\]

2. **Reading Address Circuit** \(\{C_j^w\}_{j \in [\theta]}\): First, for \(j \in [\theta]\), homomorphically compute ciphertext \(\text{Step}[i].\text{homct}_j^w\) with respect to \(C_j^w\). Then use the recoding key \(\text{Step}[i].\text{rk}_{w}^a\) to recode the read address information of \(i\)-th step to \((i+1)\)-th step. Next, homomorphically evaluate the gadget circuit \(C\) and check circuit \(C^ck\) to obtain \(\text{Step}[i].ct_{w}^a\) and \(\text{Step}[i].ct_{w}^c\) respectively. Recode the value residing in correct location of database using \(\text{Step}[i].\text{rk}_{k\ell}^w\), where \(k, \ell\) can be determined by the execution of \(P^{D}\). The detail follows: For \(j \in [\theta]\), evaluate \(C_j^w\) as

\[
\text{Step}[i].\text{homct}_j^w \leftarrow \text{CHR.CtEval}((\{\text{Step}[i].ct_{st}^w\}_{k \in [\tau]}, \{\text{Step}[i].ct_{ra}^w\}_{k \in [\theta]}, \text{Step}[i].ct, C_j^w)\)
\]

Then for \(j \in [\theta]\), compute \(\text{Step}[i+1].ct_{w}^a = \text{CHR.ReEnc}(c_0, \text{Step}[i].\text{homct}_j^w, \text{Step}[i].\text{rk}_{w}^a)\). Next evaluate gadget circuit \(C\) and \(C^ck\) (c.f. Figure 6)

\[
\text{Step}[i].ct_{w}^c = \text{CHR.CtEval}(\{\text{Step}[i].\text{homct}_j^w\}_{j \in [\theta]}, C)\)
\]

where \(\text{Step}[i].ct_{w}^a\) is encoding of read address \(j\) and \(\text{Step}[i].ct_{w}^c\) is encoding of last written time \(k\), and \(u_i\) is the encoding of time-step \(i\) that can be obtained by computing \(u_i = \text{CHR.ReEnc}(c_0, u_{i-1}, \text{Step}[i].\text{rk}_i^w)\). We can determine \(k \in [N]\) and \(\ell \in [i-1]\) by executing \(P\) on the database \(D\). And choose the corresponding re-key \(\text{Step}[i].\text{rk}_{k\ell}^w\), then compute

\[
\text{Step}[i+1].ct' = \text{CHR.ReEnc}(c_0, \text{Step}[i].ct', \text{Step}[i].ct_{w}^c, \text{Step}[\ell].ct_{k\ell}^{db}, \text{Step}[i].\text{rk}_{k\ell}^w)
\]

3. **Writing Address/Value Circuits** \(\{C_j^w\}_{j \in [\theta]}, C^{\text{wb}}\): First, for \(j \in [\theta]\), homomorphically compute ciphertext \(\text{Step}[i].\text{homct}_j^w\) with respect to \(C_j^w\) and compute \(\text{Step}[i].ct_{w}^{\text{wb}}\) with respect to \(C^{\text{wb}}\). Then, evaluate the gadget circuit \(C\) on input ciphertexts \(\{\text{Step}[i].\text{homct}_j^w\}\) to obtain \(\text{Step}[i].ct_{w}^{\text{wb}}\). Next, recode the writing value along with the writing address to \(\text{Step}[i].ct_{k\ell}^{db}\), using recoding key \(\text{Step}[i].\text{rk}_{w}^{db}\), where \(k\) can be determined by the execution of \(P^{D}\). The detail follows: For \(j \in [\theta]\), evaluate \(C_j^w\) and \(C^{\text{wb}}\) as

\[
\text{Step}[i].\text{homct}_j^w \leftarrow \text{CHR.CtEval}((\{\text{Step}[i].ct_{st}^w\}_{k \in [\tau]}, \{\text{Step}[i].ct_{ra}^w\}_{k \in [\theta]}, \text{Step}[i].ct, C_j^w)\)
\]

\[
\text{Step}[i].ct_{w}^{\text{wb}} \leftarrow \text{CHR.CtEval}((\{\text{Step}[i].ct_{st}^w\}_{k \in [\tau]}, \{\text{Step}[i].ct_{ra}^w\}_{k \in [\theta]}, \text{Step}[i].ct, C^{\text{wb}})\)
\]

Then evaluate \(\text{Step}[i].ct_{w}^{\text{wb}} = \text{CHR.CtEval}(\{\text{Step}[i].\text{homct}_j^w\}_{j \in [\theta]}, \text{C})\). Next, pick the corresponding recoding key \(\text{Step}[i].\text{rk}_{w}^{db}\) where \(k\) is the writing address, and compute

\[
ct_{k\ell}^{db} = \text{CHR.ReEnc}(c_0, \text{Step}[i].ct_{w}^{\text{wb}}, \text{Step}[i].\text{rk}_{w}^{db})
\]

\textit{For ease of notation, we omit the public keys in the input to algorithm CHR.ReEnc when the context is clear.}
4. Update circuit \( \{C_{j}^{up}\}_{j \in ([i+1] \eta]} \): First, for \( j \in ([i+1] \eta] \), homomorphically compute ciphertext \( \text{Step}[i].\homoc_{j}^{lt} \) with respect to \( C_{j}^{up} \). Then use the recoding key \( \text{Step}[i].r_{k_{j}}^{lt} \) to recode the address list information of \( i \)-th step to \( (i+1) \)-th step. The detail follows: For \( j \in ([i+1] \eta] \), evaluate the update circuit \( C_{j}^{up} \) (c.f. Figure 5) as

\[
\text{Step}[i].\homoc_{j}^{lt} \leftarrow \text{CHR}.\text{CtEval}(\{\text{Step}[i].\ct_{j}^{lt}\}_{t \in [i \eta]}, \text{Step}[i].\ct_{j}^{w}, u_{i}, C_{j}^{up})
\]

Then for \( j \in ([i+1] \eta] \), compute

\[
\text{Step}[i + 1].\ct_{j}^{lt} = \text{CHR}.\text{ReEnc}(c_{0}, \text{Step}[i].r_{k_{j}}^{lt}, \text{Step}[i].\homoc_{j}^{lt})
\]

The algorithm of last step is to compute \( \text{CHR}.\text{EqTest}(pk_{out}, \text{Step}[t].\ct_{out}^{\psi}, \psi) \), where \( t \) denotes the time step when \( P^{D} \) halts. If output of \( \text{CHR}.\text{EqTest} \) is equal, then output 0; otherwise output 1.

We show that the above scheme is a secure ABE for RAMs scheme. In particular, we prove the following theorem.

**Theorem 5.1.** Assuming CHR for a class of controlled functions \( F \) (as defined in Equation (6)), ABE (described above) is a secure ABE for RAMs scheme.

### 5.1 Correctness and Efficiency Analysis

We now show that the above ABE scheme satisfies the properties of correctness and desired efficiency.

**Correctness.** We show the correctness proof below.

**Lemma 5.2.** Assuming the correctness of CHR for the set of controlled functions \( F \) as defined in Equation (6), the above ABE construction satisfies correctness as defined in Definition 2.1.

**Proof.** We first define a notion, named \( i \)-th step temporary key

\[
\text{Step}[i].\text{TempKey} = (\{\text{Step}[i].pk_{k}^{lt}\}_{k \in [r]}, \{\text{Step}[i].pk_{k}^{\eta}\}_{k \in [\eta]}, \text{Step}[i].pk)
\]

Let the ciphertext be \( \ct_{D} \) and secret key be \( sk_{P} \). At \( i \)-th step, by evaluating the ciphertext with respect to circuits \( \{C_{j}^{st}\}_{j \in [r]}, \{C_{j}^{t}\}_{j \in [\eta]}, \{C_{j}^{w}\}_{j \in [\eta]}, C^{wb} \) and auxiliary circuits \( C^{ck}, \{C_{j}^{up}\}_{j \in ([i+1] \eta]} \), we obtain the homomorphic ciphertexts

\[
\{\text{Step}[i].\homoc_{j}^{st}\}_{j \in [r]}, \{\text{Step}[i].\homoc_{j}^{t}\}_{j \in [\eta]}, \text{Step}[i].\ct_{j}^{w}, \{\text{Step}[i].\homoc_{j}^{lt}\}_{j \in ([i+1] \eta]}
\]

By the correctness of CHR scheme (c.f. Definition 3.1), we have that ciphertexts \( \text{Step}[i+1].\ct_{j}^{st}, \text{Step}[i+1].\ct_{j}^{t}, \text{Step}[i+1].\ct_{j}^{lt} \) encrypt the same message under \( \text{Step}[i+1].\text{TempKey} \) as their \( i \)-th step homomorphically evaluated ciphertexts respectively. Since the evaluation of RAM program \( P \) on database \( D \) is in the clear, so in the reading part, we choose the correct recoding key \( \text{Step}[i].r_{k_{j}}^{lt} \) to recode the value in the \( k \)-th location to \( \text{Step}[i+1]\ct_{j}^{lt} \).

Suppose at step \( t \), where \( t \leq T \), we have \( P^{D} = 0 \), then evaluate

\[
\text{Step}[t].\ct^{st} = \text{CHR}.\text{CtEval}(\{\text{Step}[t].\homoc_{j}^{st}\}_{j \in [r]}, C)
\]

where \( \text{Step}[t].\ct^{st} \) encrypts 0. Again by correctness of CHR scheme, the re-encrypted ciphertext \( \text{Step}[t].\ct_{out}^{\psi} \) is also an encryption of 0. At last, the equality test \( \text{CHR}.\text{EqTest}(pk_{out}, \text{Step}[t].\ct_{out}^{\psi}, \psi) \) outputs correct \( \mu \) by the property of algorithm \( \text{CHR}.\text{EqTest} \).  

\[\blacksquare\]
Remark 5.3. As mentioned in Remark 2.2, our ABE construction can support auxiliary input $y$, i.e. $P^D(y)$, where this additional input $y$ serves as initial input of step circuit. The only change in the current construction is that in the encryption algorithm, we encode $y$ as $ct^{aux} = \text{CHR.} \text{Enc}(pk^{aux}, y, s)$, where $pk^{aux}$ denotes public key for auxiliary input. The correctness and security proofs closely follow the current ones.

Lattice Parameters Setting. Our ABE construction is based on CHR for a class of controlled functions $F$ (as defined in Equation (6), which may have other instantiations besides the lattice-based one as specified in this paper. To make our ABE construction more clear from the lattice-based perspective, we here set the lattice parameters for the underlying CHR scheme. As mentioned before, in the CHR scheme, if the step circuit being evaluated has depth $d$, the noise in the ciphertexts grows in the worst case by a factor of $O(m^d)$. Thus, to support a RAM program with worst-case running time $T$ (the unit of time corresponds to one step), we set the $(n, m, q)$ as

- Lattice dimension $n$ is an integer such that $n \geq (Td \log n)^{1/\epsilon}$, for some fixed $0 < \epsilon < 1/2$.
- Modulus $q$ is set to be $q = 2^{n^\epsilon}$, since the noise in the ciphertexts grows by a factor of $O(m^T d)$. Hence, we need $q$ to be on the order of $\Omega(Bm^T d)$, where $B = O(n)$ is the maximum magnitude of noise (from discrete Gaussian distribution) added during encryption. To ensure correctness of decryption and hardness of LWE, we set $q = 2^{n^\epsilon}$.
- Lattice column parameter $m$ is set to be $m = \Theta(n \log q)$ to make the leftover hash lemma hold.

For security we rely on the hardness of the LWE problem, which requires that the ratio $q/B$ is not too large, where $B = O(n)$ is the maximum magnitude of noise (from discrete Gaussian distribution) added during encryption. In particular, the underlying problem is believed to be hard even when $q/B$ is $2^{n^\epsilon}$.

Complexity of Our Construction. The (space/time) complexity of our construction can be analyzed by the following aspects. Here, we use the notations in Table 3 and our lattice-based CHR instantiation. The polynomial $n(\cdot, \cdot)$ denotes the lattice dimension.

- The public parameters contain $(\tau + \theta + \eta + N + 3) n \times m$ matrices in $\mathbb{Z}_q$, which is $\tilde{O}(n(\lambda,T)^2 \cdot NT)$ in bit complexity. The master secret key is one $m \times m$ matrix.
- The secret key for RAM program $P$ contains $T (NT(T+1)/2 + \eta(T+2)T/2 + 2 + \tau + \theta)$ $m \times m$ small matrices, which is $\tilde{O}(n(\lambda,T)^2 \cdot NT^3)$ in bit complexity.
- The ciphertext for database with length $N$ contains $(3 + \tau + \theta + \eta + N)$ dimension-$m$ vectors in $\mathbb{Z}_q$, which is $\tilde{O}(n(\lambda,T)^2 \cdot NT)$ in bit complexity.
- Suppose the running time of $P^D$ is $t$, and decryption involves matrix-vector multiplication. The time complexity of decryption is $\tilde{O}(n(\lambda,T)^2 \cdot t)$.

Sub-linear Decryption. We would like to show the following: if a program $P$ on input database $D$ takes time at most $T$ then correspondingly, the decryption of secret key for $P$ on input an encryption of message $x$ associated with attribute database $D$ takes time $p(\lambda,T)$, for a fixed polynomial $p$.

We analyze the time to decrypt an encryption of database $D$ associated with message $x$ using a key of RAM program with runtime bounded by $T$: observe that in the description of ABE.Dec,
bullets 1, 2, 3 and 4 are executed $T$ number of steps. We focus on bounding the running time of bullets 1, 2, 3 and 4 in any given step. We analyze all four cases below.

- **State circuit**: The runtime of $\text{CHR.CtEval}$ is a polynomial in $(\lambda, \tau, \theta, T\eta)$. Observe that $\tau$ is the length of the state, which is independent of the input length, and $\theta = \log N, \eta = \log T + \log N$. Thus, the runtime of $\text{CHR.CtEval}$ is upper bounded by a polynomial in $(\lambda, T)$. The runtime of $\text{CHR.ReEnc}$ is bounded by a polynomial in $(\lambda, \tau)$.

- **Reading address circuit**: In this step, $\text{CHR.CtEval}$ is executed twice. The runtime of first execution of $\text{CHR.CtEval}$ is a polynomial in $(\lambda, \theta)$. The runtime of $\text{CHR.CtEval}$ is upper bounded by $(\lambda, T\eta)$. Determining $j$ and $k$ takes time at most $T$. The runtime of $\text{CHR.ReEnc}$ is bounded by a polynomial in $\lambda$.

- **Writing address/value circuits**: In this step, $\text{CHR.CtEval}$ is executed twice. In both executions, the runtime is bounded by a polynomial in $(\lambda, \tau, \theta, T\eta)$. The runtime of $\text{CHR.ReEnc}$ is bounded by a polynomial in $\lambda$.

- **Update circuit**: In this step, $\text{CHR.CtEval}$ is bounded by a polynomial in $(\lambda, T\eta)$. In this step, the runtime of $\text{CHR.ReEnc}$ is upper bounded by $(\lambda, T\eta)$. The runtime of $\text{CHR.ReEnc}$ is upper bounded by $(\lambda)$.

From the above observations, it follows that the runtime of the decryption algorithm is a polynomial in $(\lambda, T)$, where the polynomial is independent of the length of the database.

In particular, notice that if $T$ is polylogarithmic in the input length then the decryption time is sub-linear in the input length.

### 5.2 Security Proof

We prove the security of our ABE construction based on security of controlled homomorphic recoding scheme. Before proceeding to the proof, we describe some auxiliary algorithms that are useful to the proof. There are five algorithms:

- **Sim.ABESetup** produces “programmed” public keys. That is, every public key produced as part of setup has hardwired in it, a bit of the challenge database. To perform this operation, we invoke the indistinguishability of setup security of $\text{CHR}$ (Definition 3.2).

- **Sim.StepKey** takes as input the $i$-th layer of simulated public keys (called temporary keys below) and produces the $i$-th layer of simulated recoding keys and $(i+1)$-th layer of simulated public keys, except for terminating keys $\text{Step}[i].rk_{\text{out}}$, which is used for recoding from current step to final step is the program terminates. To perform this operation, we invoke the indistinguishability of simulated keys (Definition 3.3).

- **Sim.OutKey** takes as input the $i$-th layer of simulated public keys and produces the simulated terminating keys. We use that fact $D_i^{D^*} \neq 0$ for any queried program $P_i$ to simulate the terminating keys $\text{Step}[i].rk_{\text{out}}$. To perform this operation, we invoke the indistinguishability of recoding keys (Definition 3.4).

- **Real.StepKey** on input the $i$-th layer of simulated public keys (called temporary keys) and master secret key of $\text{CHR}$, it produces the $i$-th layer of ‘real’ recoding keys and $(i+1)$-th layer of ‘real’ public keys.

- **Sim.Enc** produces a simulated encryption of the message. To perform this, we invoke the pseudorandomness of ciphertext property (Definition 3.5).
Proof Intuition. We explain the intuition of the proof next. For explaining the intuition, we focus on weak selective security, where the adversary submits all the queries in the very beginning of the security experiment. The adversary \( A \) submits the database \( D^* \), secret message \( \mu \) and program queries \( P_1, \ldots, P_Q \) such that \( P_1^{D^*} \neq 0 \).

Hybrid \( H_1 \) corresponds to the real experiment, where all the parameters are sampled according to \( CHR \). First, the challenger simulates the public keys of \( ABE \) using the algorithm Sim.ABESetup on input the database \( D^* \) (hybrid \( H_2 \)). Also, in the same hybrid, generate all the layers of the recoding keys in every attribute key using \( \text{Real.StepKey} \). The challenger, over a sequence of hybrids \( (H_{3,1,\lambda}) \), starts manipulating the attribute key of \( P_1 \). Recall that the attribute key of \( P_1 \) consists of \( T \) sets of recoding keys, the step-keys. Next, switch every intermediate layer of recoding layers to be simulated. That is, in the \( j \)-th step \( (H_{3,1,j}) \), all the \( (j-1) \) layers of recoding keys are simulated using Sim.StepKey while all the layer from \( (j+1) \)-th onwards are computed using \( \text{Real.StepKey} \). Switch the \( j \)-th layer of recoding keys to be simulated using Sim.StepKey. At the end of this sequence of hybrids, all the layers of recoding keys in the attribute key of \( P_1 \) are simulated. Perform this sequence of hybrids to the rest of the attribute keys associated with \( P_2, \ldots, P_Q \). Once this is done, simulate the ciphertext of \( \mu \) using Sim.Enc.

We present a formal descriptions of the above algorithms.

- \( \text{Sim.ABESetup}(1^\lambda, D^*) \): On input security parameter \( \lambda \) and challenge database \( D^* = \{ x^*_i \}_{i=1}^N \), the simulated setup algorithm first generate the anchor public key \( pk_0 \) along with \( sk_0 \) by running \( (pk_0, sk_0) \leftarrow \text{CHR.Setup}(1^\lambda) \) and simulate public key for 0-th time step \( pk_{0}^{\text{out}} \) as

\[
(\text{Simpk}^0_t, \text{Simpk}_{\text{out}}) \leftarrow \text{Sim.GenCHRSetup}(1^\lambda, 0, 2; pk_0)
\]

Then simulate the rest of public parameters as

1. Simulate public keys \( \text{Step}[0].\text{Simpk}^t \) for initial state as

\[
\{\text{Step}[0].\text{Simpk}^t\}_{\lambda \in [\tau]} \leftarrow \text{Sim.GenCHRSetup}(1^\lambda, 1, \tau; pk_0)
\]

2. Embed the challenge attribute database \( D^* \) into \( \text{Step}[0].\text{Simpk}^d \) as

\[
\{\text{Step}[0].\text{Simpk}^d\}_{\lambda \in [N]} \leftarrow \text{Sim.GenCHRSetup}(1^\lambda, D^*, N; pk_0)
\]

3. Simulate public keys \( \text{Step}[0].\text{Simpk}^r \) for read address as \( \text{Step}[0].\text{Simpk}^r \leftarrow \text{Sim.GenCHRSetup}(1^\lambda, 1, 1; pk_0) \)

\[
\{\text{Step}[0].\text{Simpk}^r\}_{\lambda = 2} \leftarrow \text{Sim.GenCHRSetup}(1^\lambda, 0, \theta - 1; pk_0)
\]

4. Simulate public keys \( \text{Step}[0].\text{Simpk}^l \) for address list as

\[
\{\text{Step}[0].\text{Simpk}^l\}_{\lambda \in [\eta]} \leftarrow \text{Sim.GenCHRSetup}(1^\lambda, 0^\eta, \eta; pk_0)
\]

Output master secret key \( msk = sk_0 \) and simulated public parameter \( \text{Sim.pp} \) as

\[
\text{Sim.pp} = (\{\text{Step}[0].\text{Simpk}^t\}_{\lambda \in [\tau]}, \{\text{Step}[0].\text{Simpk}^r\}_{\lambda \in [\theta]}, \{\text{Step}[0].\text{Simpk}^l\}_{\lambda \in [\eta]}, \{\text{Step}[0].\text{Simpk}^d\}_{\lambda \in [N]}, \text{Simpk}^0_t, pk_0, \text{Simpk}_{\text{out}})
\]

- \( \text{Sim.StepKey}(\text{Sim.pp}, msk, \{\text{Step}[j].\text{SimTempKey}\}_{j \in [\ell]}, P^{D^*}) \): On input simulated public parameters \( \text{Sim.pp} \), master secret key \( msk \), temporary keys \( \{\text{Step}[i].\text{SimTempKey}\}_{j \in [\ell]} \) and \( i \)-th step internal state \( y_i \) (including state, reading address, etc., obtained by executing a RAM program
on challenge database $D^*$ upto the $i$-th step), the simulated step key generation outputs $i$-th step key $\text{Step}[i].\text{SimKey}$ and $(i + 1)$-th simulated temporary key $\text{Step}[i + 1].\text{SimTempKey}$. For $j \in [i]$, the algorithm parses

\[
\text{Step}[j].\text{SimTempKey} = (\text{Step}[j].\text{Simpk}^j, \{(\text{Step}[j].\text{Simpk}_{k}^j)_{k \in [\tau]}\}, \{(\text{Step}[j].\text{Simpk}_{k}^a)_{k \in [\theta]}\},
\{\text{Step}[j].\text{Simpk}_{k}^{db}\}_{k \in [N]}, \{\text{Step}[j].\text{Simpk}_{k}^{lt}\}_{k \in (j + 1)\eta}, \text{Step}[j].\text{Simpk})
\]

We use $\text{Step}[j].\text{SimTrdr}$ to denote the trapdoor of $\text{Step}[j].\text{SimTrdr}$. For the $i$-th step of execution $P^{D^*}$, let $\{\text{st}_{ik}\}_{k \in [\tau]}$ be the internal state, $\{\text{ra}_{ik}\}_{k \in [\theta]}$ be the reading address, $u_i$ be the writing address, $wb_i$ be the writing bit, and $\{\text{lt}_{ik}\}_{k \in (i + 1)\eta}$ be the update list. Denote the $i$-th step execution status $E_i$ for $P^{D^*}$ as

\[
E_i = (\{\text{st}_{ik}\}_{k \in [\tau]}, \{\text{ra}_{ik}\}_{k \in [\theta]}, \text{rb}_i)
\]

Output $\bot$ if $\text{Step}[i].\text{rk}^{\text{out}}$ if $f(\{C_j^{\text{st}}(E_i)\}_{j = 1}^\tau) = 1$, where controlled function $f$ is defined in Definition $(7)$. Otherwise, evaluate the public keys as

\[
\text{Step}[i].\text{Simpk}_{j}^{st} \leftarrow \text{CHR.KeyEval}(\{\text{Step}[i].\text{Simpk}_{k}^{st}\}_{k \in [\tau]}, \{\text{Step}[i].\text{Simpk}_{k}^{a}\}_{k \in [\theta]}, \text{Step}[i].\text{Simpk}, C_{j}^{st})
\]

And compute $\text{Step}[i].\text{Simpk}_{j}^{st} = \text{CHR.KeyEval}(\{\text{Step}[i].\text{Simpk}_{j}^{st}\}_{j \in [\tau]}, C)$, then use the secret key $\text{sk}_0$ to compute the reencoding key as

\[
\text{Step}[i].\text{rk}^{\text{out}} \leftarrow \text{CHR.ReEncKG}(\text{pk}_0, \text{Step}[i].\text{Simpk}_{j}^{st}, \text{sk}_0, \text{pk}^{\text{out}}, h)
\]

Do the following:

1. **State circuit** $\{C_k^{st}\}_{k \in [\tau]}$: For $k \in [\tau]$, compute

\[
(\text{Step}[i].\text{Simpk}_{k}^{st}, \text{Step}[i + 1].\text{Simpk}_{k}^{st}) \leftarrow \text{Sim.CHRkey}(\text{pk}_0, \text{Step}[i].\text{SimKey}, \text{Step}[j].\text{SimTrdr}, E_i, C_k^{st}, \text{Ind})
\]

2. **Reading circuit** $\{C_k^{r}\}_{k \in [\theta]}$: For $k \in [\theta]$, compute

\[
(\text{Step}[i].\text{Simpk}_{k}^{r}, \text{Step}[i + 1].\text{Simpk}_{k}^{r}) \leftarrow \text{Sim.CHRkey}(\text{pk}_0, \text{Step}[i].\text{SimKey}, \text{Step}[j].\text{SimTrdr}, E_i, C_k^{r}, \text{Ind})
\]

3. **Writing circuit** $\{C_k^{w}\}_{k \in [\theta]}$ and $C^{wb}$: For $j \in [N]$, compute

\[
(\text{Step}[i].\text{Simpk}_{j}^{a}, \text{Step}[i + 1].\text{Simpk}_{j}^{d_b}) \leftarrow \text{Sim.CHRkey}(\text{pk}_0, \text{Step}[i].\text{SimKey}, \text{Step}[j].\text{SimTrdr}, E_i, C_k^{w}, f_j)
\]

4. **Update circuit** $\{C_k^{up}\}_{k \in (i + 1)\eta}$: For $k \in [i + 2]\eta$, compute

\[
(\text{Step}[i].\text{Simpk}_{k}^{h}, \text{Step}[i + 1].\text{Simpk}_{k}^{h}) \leftarrow \text{Sim.CHRkey}(\text{pk}_0, \text{Step}[j].\text{SimKey}, \{\text{Step}[j].\text{Simpk}_{k}^{h}\}_{k \in (j + 1)\eta}, \text{Step}[j].\text{SimTrdr}, w_i, i, \{\text{lt}_{ik}\}_{k \in (i + 1)\eta}, C_k^{h}, \text{Ind})
\]

5. **Time step**: Compute

\[
(\text{Step}[i].\text{Simpk}^t, \text{Step}[i + 1].\text{Simpk}^t) \leftarrow \text{Sim.CHRkey}(\text{pk}_0, \text{Step}[j].\text{Simpk}^t, \text{Step}[j].\text{SimTrdr}, g_i)
\]

Let circuit $C^{ra} = C(\{C_k^{ra}\}_{k \in [\theta]})$ and calculate $C^{ra}(y_i) = j^*, C^{up}(y_i, 0) = k^*$. Then compute

\[
(\{\text{Step}[i].\text{Simpk}^j_{jk}\}_{j \in [N], k \in [i - 1]}, \text{Step}[i + 1].\text{Simpk}) \leftarrow \text{Sim.GenCHRkey}(\text{pk}_0, \text{Step}[i].\text{SimTempKey}, \text{Step}[j].\text{SimTrdr}, E_i, C^{ra}, C^{up}, N, i - 1)
\]
Output \((E_{i+1}, \text{Step}[i], \text{SimKEY}, \text{Step}[i+1], \text{SimTempKey})\) as

\[
\text{Step}[i].\text{SimKEY} = \{ \text{Step}[i].\text{rk}^\text{out}, \text{Step}[i].\text{rk}^\ell, \{ \text{Step}[i].\text{Simrk}^{\text{st}}_j \}, \{ \text{Step}[i].\text{Simrk}^{\text{ta}}_j \}, \\
\{ \text{Step}[i].\text{Simrk}^{\text{sb}}_j \}, \{ \text{Step}[i].\text{Simrk}^b_j \}, \{ \text{Step}[i].\text{Simrk}^{\text{tb}}_j \}, \{ \text{Step}[i].\text{Simrk}^{\text{tt}}_j \} \}
\]

\[
\text{Step}[i+1].\text{SimTempKey} = \{ \text{Step}[i+1].\text{Simp}^{\text{out}}_i, \{ \{ \text{Step}[j].\text{Simpk}^{\text{st}}_k \}_{j \in \llbracket r \rrbracket}, \{ \{ \text{Step}[j].\text{Simpk}^{\text{ta}}_k \}_{j \in \llbracket \theta \rrbracket}, \\
\{ \text{Step}[j].\text{Simpk}^{\text{sb}}_k \}_{j \in \llbracket N \rrbracket}, \{ \text{Step}[j].\text{Simpk}^b_k \}_{j \in \llbracket N \rrbracket}, \{ \text{Step}[j].\text{Simpk}^{\text{tb}}_k \}_{j \in \llbracket N \rrbracket}, \{ \text{Step}[j].\text{Simpk}^{\text{tt}}_k \}_{j \in \llbracket N \rrbracket} \} \}
\]

- **Sim.OutKey(Sim.pp, \{ \text{Step}[j].\text{SimTempKey} \}_{j \in \llbracket i \rrbracket}, P^{D^*})**: On input simulated public parameters Sim.pp, temporary keys \{ \text{Step}[j].\text{SimTempKey} \}_{j \in \llbracket i \rrbracket} and \(i\)-th step internal state \(y_i\) (including state, reading address, etc.), obtained by executing a RAM program on challenge database \(D^*\) upto the \(i\)-th step), the simulated out key generation outputs \(i\)-th simulated out key \(\text{Step}[i].\text{Simp}^{\text{out}}_i\). Denote the \(i\)-th step execution status \(E_i\) for \(P^{D^*}\) as above. Output \(\perp\) if \(f(\{ C^{\text{st}}_j(E_i) \}_{j \in \llbracket i \rrbracket}) = 1\). Otherwise, compute and output

\[
\text{Step}[i].\text{Simp}^{\text{out}}_i \leftarrow \text{Sim.GenCHR}_{rk}(pk_0, \text{Step}[i].\text{SimTempKey}, E_i, \{ C^{\text{st}}_k \}_{k \in \llbracket 1, \theta \rrbracket})
\]

- **Real.StepKey(Sim.pp, msk, \{ \text{Step}[j].\text{SimTempKey} \}_{j \in \llbracket i \rrbracket})**: On input simulated public parameters Sim.pp, simulated temporary keys \{ \text{Step}[j].\text{SimTempKey} \}_{j \in \llbracket i \rrbracket} and master secret key msk, the real step key generation outputs \(i\)-th step key \(\text{Step}[i].\text{KEY}\) and \((i+1)\)-th temporary key \(\text{Step}[i+1].\text{TempKey}\). The algorithm parses

\[
\text{Step}[i].\text{SimTempKey} = \{ \{ \text{Sim.}\text{Step}[i].\text{pk}^{\text{st}}_j \}_{j \in \llbracket r \rrbracket}, \{ \{ \text{Sim.}\text{Step}[i].\text{pk}^{\text{ta}}_j \}_{j \in \llbracket \theta \rrbracket}, \{ \text{Sim.}\text{Step}[i].\text{pk}^{\text{sb}}_j \}_{j \in \llbracket N \rrbracket}, \\
\{ \text{Sim.}\text{Step}[i+1].\text{pk}^b_j \}_{j \in \llbracket N \rrbracket}, \text{Sim.}\text{Step}[i].\text{pk} \}
\]

Output \(\perp\) if \(\text{Step}[i].\text{rk}^\ell\) if \(f(\{ C^{\text{st}}_j(E_i) \}_{j \in \llbracket i \rrbracket}) = 1\), where controlled function \(f\) is defined in Definition (7). Otherwise, set \(C^{\text{st}} = C(\{ C^{\text{st}}_j \}_{j \in \llbracket r \rrbracket})\) and compute

\[
\text{Step}[i].\text{rk}^\ell \leftarrow \text{DerivReKey}(pk_0, \text{Step}[i].\text{SimTempKey}, y_i, \text{Sim.p}^{\text{out}}_i, msk, C^{\text{st}})
\]

For time step, first generate \(\text{Step}[i+1].\text{pk}^\ell\) using CHR.Setup(1\(^\text{st}\)), and then compute

\[
\text{Step}[i].\text{rk}^\ell \leftarrow \text{CHR.\text{ReEncKG}}(pk_0, \text{Step}[i].\text{Simpk}^\ell, sk_0, \text{Step}[i+1].\text{pk}^\ell, g_i)
\]

Then for \(h \in \llbracket N \rrbracket, j \in \llbracket r \rrbracket, k \in \llbracket \theta \rrbracket, \ell \in \llbracket (i+1)\eta \rrbracket\), generate \(\text{Step}[i].\text{pk}^{\text{sb}}_h, \text{Step}[i+1].\text{pk}^{\text{st}}_j, \text{Step}[i+1].\text{pk}^{\text{ta}}_j, \text{Step}[i+1].\text{pk}^b_k, \text{Step}[i+1].\text{pk}^\ell_{\theta}, \text{Step}[i+1].\text{pk}^\ell_\ell, \text{Step}[i+1].\text{pk}^\ell_{\theta}\) using CHR.Setup(1\(^\text{st}\)) and then execute

\[
\text{Step}[i].\text{rk}^{\text{st}}_j \leftarrow \text{DerivReKey}(pk_0, \text{Step}[i].\text{SimTempKey}, msk, C^{\text{st}}_j, \text{Step}[i+1].\text{pk}^{\text{st}}_j, \text{Ind})
\]

\[
\text{Step}[i].\text{rk}^{\text{ta}}_k \leftarrow \text{DerivReKey}(pk_0, \text{Step}[i].\text{SimTempKey}, msk, C^{\text{ta}}_k, \text{Step}[i+1].\text{pk}^{\text{ta}}_k, \text{Ind})
\]

\[
\text{Step}[i].\text{rk}^{\text{sb}}_\ell \leftarrow \text{DerivReKey}(pk_0, \text{Step}[i].\text{SimTempKey}, msk, C^{\text{sb}}_\ell, \text{Step}[i+1].\text{pk}^{\text{sb}}_\ell, f_h)
\]

\[
\text{Step}[i].\text{rk}^{\ell}_{\theta} \leftarrow \text{DerivReKey}(pk_0, \text{Step}[i].\text{SimTempKey}, msk, C^{\ell}_{\theta}, \text{Step}[i+1].\text{pk}^{\ell}_{\theta}, \text{Ind})
\]

where circuit \(C^{\ell} = C(\{ C^{\ell}_h \}_{h \in \llbracket \theta \rrbracket})\). Next, for \(k \in \{ i - 1 \}, j \in \llbracket N \rrbracket\), compute the following

\[
\text{Step}[i].\text{rk}^{\ell}_{kj} \leftarrow \text{DerivReKey}(pk_0, \text{Step}[i].\text{SimTempKey}, msk, C^{\ell}, C^{\text{up}}_k, \text{Step}[k].\text{Simpk}^{\text{db}}_j, f_{kj})
\]

where circuit \(C^{\ell} = C(\{ C^{\ell}_k \}_{k \in \llbracket \theta \rrbracket})\). Output \(i\)-th step key \(\text{Step}[i].\text{KEY}\) and \((i+1)\)-th temporary key \(\text{Step}[i+1].\text{TempKey}\).
• Sim.Enc(Sim.pp, D∗, µ): On input simulated public parameters Sim.pp, challenge database D∗ = \{x^*_i\}_{i=1}^N and message µ, the simulated encryption algorithm outputs simulated ciphertext Sim.ct. For ease of notation, we use Step[0].SimTrdr to denote the all trapdoor information of step 0. The algorithm randomly chooses a secret message s, and for i ∈ [N], encrypts the database as

\[
\text{Step}[0].\text{Simct}^{db}_i = \text{Sim.CHR}_{ct}(\text{Step}[0].\text{Simpk}^{db}_i, \text{Step}[0].\text{SimTrdr}, x^*_i, s)
\]

For i ∈ [τ], encrypt the initial state as

\[
\text{Step}[0].\text{Simct}^{st}_i = \text{Sim.CHR}_{ct}(\text{Step}[0].\text{Simpk}^{st}_i, \text{Step}[0].\text{SimTrdr}, 1, s)
\]

Next, it encrypt the auxiliary information as c_0 = Sim.CHR_{ct}(Sim.pk_0, I, 0, s), u_1 = Sim.CHR_{ct}(Sim.pk_0, \text{Step}[0].\text{SimTrdr}, 0, s) and initial list as Step[0].Simct^{st}_i = Sim.CHR_{ct}(Step[0].Simpk^{st}_i, \text{Step}[0].\text{SimTrdr}, 0, s) for i ∈ [η]. Finally, choose a random vector ψ over the ciphertext space of CHR.

**Theorem 5.4.** Assuming the security of CHR for controlled funcions \(\mathcal{F}\), the scheme ABE satisfies the definition of ABE security (c.f. Definition 2.4).

**Proof.** Let \(Q\) be the number of key queries made by the adversary. We first describe a sequence of hybrids as follows:

**Hybrid H_1:** This corresponds to the real experiment.

- \(\mathcal{A}\) specifies challenge attribute database \(D^*\) and message \(\mu\).
- Challenger computes Setup(\(1^\lambda\)) to obtain the public parameters pp and secret key msk. Then challenger generates the challenge ciphertext \(ct^* \leftarrow \text{Enc}(pp, D^*, \mu)\). It sends \(ct^*\) and \(pp\) to \(\mathcal{A}\).
- For \(i \in [Q]\), adversary \(\mathcal{A}\) specifies the programs \(P_i\) such that \(P^D_i \neq 0\). Challenger generates the attribute keys for \(P_i\), for \(i \in [Q]\), \(sk_{P_i} \leftarrow \text{KeyGen}(msk, P_i)\). In more detail, \(sk_{P_i}\) is generated as follows:
  - For every \(j \in [T]\), compute
    \[
    \text{Step}[j].\text{KEY}_{i} \leftarrow \text{Real.StepKey}(pp, msk, \{\text{Step}[j].\text{TempKey}_{i}\}_{k \in [T]})
    \]
    - Set \(sk_i = (\{\text{Step}[j].\text{KEY}_{i}\}_{j \in [T]})\).
- Let \(b\) be the output of adversary. Output \(b\).

**Hybrid H_2:** \(H_2\) is the same as \(H_1\) except that it uses Sim.ABESetup(\(1^\lambda, D^*\)) to generate Sim.pp and msk.

- \(\mathcal{A}\) specifies attribute \(D^*\) and message \(\mu\).
- Challenger generates the setup Sim.ABESetup(\(1^\lambda, D^*\)) to obtain the simulated public key Sim.pp and master secret key msk. Then challenger generates the challenge ciphertext \(ct^* \leftarrow \text{Enc}(\text{Sim.pp, D^*, \mu})\). It sends \(ct^*\) and \(\text{Sim.pp}\) to \(\mathcal{A}\).
- For \(i \in [Q]\), adversary \(\mathcal{A}\) specifies the programs \(P_i\) such that \(P^D_i \neq 0\). Challenger generates the attribute keys for \(P_i\), for \(i \in [Q]\), \(sk_{P_i} \leftarrow \text{KeyGen}(msk, P_i)\). In more detail, \(sk_{P_i}\) is generated as follows:
  - For every \(j \in [T]\), compute
    \[
    \text{Step}[j].\text{KEY}_{i} \leftarrow \text{Real.StepKey}(\text{Sim.pp, msk, \{Step[k].TempKey}_{i}\}_{k \in [T]})
    \]

47
Hybrid \( \{H_{3,i^*,j^*}\}_{i^*,j^*\in[T]} \): Simply put, in hybrid H_{3,i,j}, for \( i < i^* \), the secret key for query \( P_i \) is simulated. For query \( P_{i^*} \), up to the \( j^* \)-th step, the step keys are simulated, for step \( j > j^* \), the step keys are generated normally. For query \( P_i \), where \( i > i^* \), its step keys are all generated normally. We describe it in details below:

- Adversary specifies attribute \( D^* \) and message \( \mu \).
- Challenger generates the setup \( \text{Sim.ABESetup}(1^\lambda, D^*) \) to obtain the simulated public key \( \text{Sim.pp} \) and master secret key \( \text{msk} \). Then challenger generates the challenge ciphertext \( \text{ct}^* \leftarrow \text{Enc}(\text{Sim.pp}, D^*, \mu) \). It sends \( \text{ct}^* \) and \( \text{Sim.pp} \) to \( \mathcal{A} \).
- For \( i \in [Q] \), adversary \( \mathcal{A} \) specifies the programs \( P_i \) such that \( P_i^{D^*} \neq 0 \). Challenger generates the secret key \( \text{sk}_{P_i} = (\{\text{Step}[j].\text{KEY}\}_{j\in[T]}) \) for \( P_i \), as follows:
  - For \( i < i^* \),
    1. For every \( j \in [T] \), compute
       \[
       (\text{Step}[j].\text{SimKEY}_i, \text{Step}[j+1].\text{SimTempKey}_i) \leftarrow \text{Sim.\text{StepKey}}(\text{Sim.pp}, \text{msk}, \{\text{Step}[k].\text{SimTempKey}_i\}_{k\in[j]}, P_i^{D^*})
       \]
       And then replace the \( \text{Step}[j].\text{Simrk}_{i^*}^{\text{out}} \) in \( \text{Step}[j].\text{SimKEY}_i \) using
       \[
       \text{Step}[j].\text{Simrk}_{i^*}^{\text{out}} \leftarrow \text{Sim.\text{OutKey}}(\text{Sim.pp}, \{\text{Step}[k].\text{SimTempKey}_i\}_{k\in[j]}, P_i^{D^*})
       \]
    2. Set \( \text{sk}_i = (\{\text{Step}[j].\text{SimKEY}_i\}_{j\in[T]}) \).
  - For \( i = i^* \),
    1. For \( j < j^* \), generate
       \[
       (\text{Step}[j].\text{SimKEY}_i, \text{Step}[j+1].\text{SimTempKey}_i) \leftarrow \text{Sim.\text{StepKey}}(\text{Sim.pp}, \text{msk}, \{\text{Step}[k].\text{SimTempKey}_i\}_{k\in[j]}, P_i^{D^*})
       \]
    2. For \( j = j^* \), generate
       \[
       (\text{Step}[j].\text{KEY}_i, \text{Step}[j+1].\text{TempKey}_i) \leftarrow \text{Real.\text{StepKey}}(\text{Sim.pp}, \text{msk}, \{\text{Step}[j].\text{SimTempKey}_i\}_{j\in[i]})
       \]
    3. For \( j > j^* \), generate
       \[
       \text{Step}[j].\text{KEY}_i \leftarrow \text{Real.\text{StepKey}}(\text{Sim.pp}, \{\text{Step}[k].\text{SimTempKey}_i\}_{k\in[j^*]}, \{\text{Step}[k].\text{TempKey}_i\}_{k=j^*+1}, \text{msk})
       \]
  4. Set \( \text{sk}_i = (\{\text{Step}[j].\text{SimKEY}_i\}_{j\in[T], j < j^*}, \{\text{Step}[j].\text{KEY}_i\}_{j\in[T], j \geq j^*}) \).
    - For \( i > i^* \),
      1. For every \( j \in [T] \), generate
         \[
         \text{Step}[j].\text{KEY}_i \leftarrow \text{Real.\text{StepKey}}(\text{Sim.pp}, \text{msk}, \{\text{Step}[k].\text{TempKey}_i\}_{k\in[j]})
         \]
    2. Set \( \text{sk}_i = (\{\text{Step}[j].\text{KEY}_i\}_{j\in[T]}) \).
- Let \( b \) be the output of adversary. Output \( b \).

\( ^{8} \)The subscript of \( \text{Step}[j].\text{KEY}_i \), \( i \) here, denotes for the \( i \)-th key query.
Hybrid \( \{ H_{3,i^*,j^*} \}_{i^*,j^* \in [T]} \): Simply put, hybrid \( \tilde{H}_{3,i^*,j^*} \) happens right after \( H_{3,i^*,j^*} \), and the only difference between these two consecutive hybrids is in \( H_{3,i^*,j^*} \), the recoding key \( \text{Step}[j^*].\text{Simrk}_i^{\text{out}} \) in \( \text{Step}[j^*] \). \( \text{SimKEY}_i \) is generated using algorithm \( \text{Sim.OutKey} \) instead of using \( \text{Sim.StepKey} \) (in hybrid \( H_{3,i^*,j^*} \)).

**Hybrid** \( H_4 \): In \( H_4 \), the secret keys for all queries are simulated without using \( \text{msk} \). Therefore, we sample the anchor public key \( \text{Sim.p} \) (with its trapdoor \( I \)) randomly from space \( \mathbb{Z}_q^{n \times m} \).

- Adversary specifies attribute \( D^* \) and message \( \mu \).
- Challenger generates the setup \( \text{Sim.ABESetup}(1^\lambda, D^*) \) to obtain the simulated public key \( \text{Sim.pp} \). Then challenger generates the challenge ciphertext \( \text{ct}^* \leftarrow \text{Enc} (\text{Sim.pp}, D^*, \mu) \). It sends \( \text{ct}^* \) and \( \text{Sim.pp} \) to \( A \).
- For \( i \in [Q] \), adversary \( A \) specifies the programs \( P_i \) such that \( P_i^{D^*} \neq 0 \). Challenger generates the attribute keys for \( P_i \), for \( i \in [Q] \). In more detail, \( \text{sk}_P_i \) is generated as follows:
  1. For every \( j \in [T] \), generate
    \[
    (\text{Step}[j].\text{SimKEY}_i, \text{Step}[j+1].\text{SimTempKey}_i) \leftarrow \text{Sim.StepKey}(\text{Sim.pp}, \{ \text{Step}[k].\text{SimTempKey}_i \}_{k \in [j]})
    \]
  2. Set \( \text{sk}_i = (\{ \text{Step}[j].\text{SimKEY}_i \}_{j \in [T]} \).
- Let \( b \) be the output of adversary. Output \( b \).

**Hybrid** \( H_5 \): \( H_5 \) is the same as \( H_4 \) except that it simulates challenge ciphertext.

- Adversary specifies attribute \( D^* \) and message \( \mu \).
- Challenger generates the setup \( \text{Sim.ABESetup}(1^\lambda) \) to obtain the simulated public key \( \text{Sim.pp} \). It sends \( \text{Sim.pp} \) to \( A \).
- Challenger generates the simulated ciphertext \( \text{Sim.ct}^* \leftarrow \text{Sim.Enc} (\text{pp}, D^*, \mu) \). It sends \( \text{Sim.ct}^* \) to \( A \).
- For \( i \in [Q] \), adversary \( A \) specifies the programs \( P_i \) such that \( P_i^{D^*} \neq 0 \). Challenger generates the attribute keys for \( P_i \), for \( i \in [Q] \), \( \text{sk}_P_i \leftarrow \text{KeyGen} (\text{msk}, P_i) \). In more detail, \( \text{sk}_P_i \) is generated as follows:
  1. For every \( j \in [T] \), generate
    \[
    (\text{Step}[j].\text{SimKEY}_i, \text{Step}[j+1].\text{SimTempKey}_i) \leftarrow \text{Sim.StepKey}(\text{Sim.pp}, \{ \text{Step}[k].\text{SimTempKey}_i \}_{k \in [j]})
    \]
  2. Set \( \text{sk}_i = (\{ \text{Step}[j].\text{SimKEY}_i \}_{j \in [T]} \).
- Let \( b \) be the output of adversary. Output \( b \).

**Lemma 5.5.** By the indistinguishability of setup property of CHR scheme (c.f. Definition 3.2), we have \( H_1 \approx H_2 \).

**Proof.** The only difference between hybrid \( H_1 \) and \( H_2 \) is that in \( H_2 \), the public parameters generated by \( \text{Sim.ABESetup}(1^\lambda, D^*) \) as described above. By the indistinguishability of setup property of CHR, the distribution \( \{ \text{Sim.pp} \} \) is statistically close to \( \{ \text{pp} \} \). Therefore, we have \( H_1 \approx H_2 \). 

**Lemma 5.6.** The output distributions of hybrids \( H_2 \) and \( H_{3,1,0} \) are identical.
Proof. As described above, hybrid $H_{3,1,0}$ is obtained by simulating $\{\text{Step}[0].\text{SimKEY}_1\}$ and generating all other step keys normally. As $\{\text{Step}[0].\text{SimKEY}_1\}$ does not exists in $sk_1$, therefore we have that hybrids $H_2$ and $H_{3,1,0}$ are identical.

\[\Box\]

**Lemma 5.7.** By the indistinguishability of simulated keys property of CHR scheme (c.f. Definition 3.3) with respect to $\mathcal{E}_{aux}$, we have $H_{3,i^*,j^*} \approx H_{3,i^*,j^*+1}$, where $j^* \in [T-1]$ and $\mathcal{E}_{aux}$ consists of two cases:

- $\mathcal{E}_{aux}$ is the distribution of simulated keys produced by Sim.CHRSsetup in the $(j^*)$th step.
- $\mathcal{E}_{aux}$ is the distribution of simulated keys produced in $1^{st}$ step, Sim.CHRSkey

Proof. The difference between $\tilde{H}_{3,i^*,j^*}$ and $H_{3,i^*,j^*+1}$ is that in $H_{3,i^*,j^*+1}$ the $(j^*+1)$-th step key $\text{Step}[j^*+1].\text{SimKEY}_{i^*}$ of query $i^*$ is simulated instead of normal generation. In algorithm $\text{Sim.StepKey}(\text{Sim.pp}, \{\text{Step}[k].\text{SimTempKey}_{i^*}\}_{k \in [j^*+1]}, \text{Step}[i].\text{Simrk}_{i^*}^\text{out}$ and the others in $\text{Step}[j^*+1].\text{SimKEY}_{i^*}$ are computed as described above. By indistinguishability of simulated keys (c.f. Definition 3.3), we have

$$\{\text{Step}[j^*+1].\text{SimKEY}_{i^*}, \text{Step}[j^*+2].\text{SimTempKey}\} \approx \{\text{Step}[j^*+1].\text{KEY}_{i^*}, \text{Step}[j^*+2].\text{TempKey}\}$$

Thus, we have $\{\text{Step}[j^*+1].\text{SimKEY}_{i^*}\} \approx \{\text{Step}[j^*+1].\text{KEY}_{i^*}\}$, which means $\tilde{H}_{3,i^*,j^*} \approx H_{3,i^*,j^*+1}$.

\[\Box\]

**Lemma 5.8.** By the indistinguishability of recoding keys property of CHR scheme (c.f. Definition 3.4) with respect to $\mathcal{E}_{aux}$, we have $H_{3,i^*,j^*} \approx \tilde{H}_{3,i^*,j^*}$, where $j^* \in [T-1]$ and $\mathcal{E}_{aux}$ is the distribution of public keys produced by Sim.CHRSsetup in the $(T-1)^{th}$ step.

Proof. The only difference between these two consecutive hybrids is in $H_{3,i^*,j^*}$, the recoding key $\text{Step}[j^*].\text{Simrk}_{i^*}^\text{out}$ in $\text{Step}[j^*].\text{SimKEY}_{i^*}^\text{in}$ is generated using algorithm $\text{Sim.OutKey}$ instead of using Sim.StepKey (in hybrid $H_{3,i^*,j^*}$). By the indistinguishability of recoding keys property of CHR scheme (c.f. Definition 3.4), the distribution of recoding key $\text{Step}[j^*].\text{Simrk}_{i^*}^\text{out}$ is computationally close to $\text{Step}[j^*].\text{rk}_{i^*}^\text{out}$, thus we have $H_{3,i^*,j^*} \approx \tilde{H}_{3,i^*,j^*}$.

\[\Box\]

**Lemma 5.9.** The output distributions of hybrids $H_{3,i^*,T}$ and $H_{3,i^*,1.0}$ are identical, when $i^* \in [Q]$.

Proof. The only difference between hybrids $H_{3,i^*,T}$ and $H_{3,i^*,1.0}$ is that in hybrid $H_{3,i^*,1.0}$, the step key $\{\text{Step}[0].\text{SimKEY}_{i^*+1}\}$ is simulated. As $\{\text{Step}[0].\text{SimKEY}_{i^*+1}\}$ does not exists in $sk_{i+1}$, thus we have that hybrids $H_{3,i^*,T}$ and $H_{3,i^*,1.0}$ are identical.

\[\Box\]

**Lemma 5.10.** The output distributions of hybrids $H_{3,Q,T}$ and $H_4$ are statistically close.

Proof. In hybrids $H_{3,Q,T}$ and $H_4$, the only difference is that in $H_{3,Q,T}$, the anchor public key $pk_0$ is generated along with $sk_0$, using algorithm $\text{TrapGen}$, while in $H_4$ the anchor public key $pk_0$ is sampled from random distribution. By Corollary 4.6, hybrids $H_{3,Q,T}$ and $H_4$ are statistically close.

\[\Box\]

**Lemma 5.11.** By the pseudorandomness of ciphertexts of CHR scheme (c.f. Definition 3.5), we have $H_4 \approx H_5$. 

50
Proof. The only difference between $H_4 \approx H_5$ is that in $H_5$ is challenge ciphertext is generated by algorithm $\text{Sim.Enc}(\text{Sim.pp}, D^*, \mu)$, where $\text{CHR.Enc}$ is used as a subroutine and a randomly chose vector $\psi$ is chosen over ciphertext space. By the pseudorandomness of ciphertexts of $\text{CHR}$, we have that the ciphertexts of both hybrids are computationally close to the uniformly random distribution over ciphertext space. Therefore, we have $H_4 \approx H_5$.

Combining the hybrids and lemmas proved above, we prove that our ABE construction is secure, as defined in Definition 2.4.

**Generic Transformation: Sub-linear Decryption to Input-Specific Runtime.** Note that the construction described above satisfies only sub-linear decryption property and in particular, does not satisfy input-specific runtime. However, [GKP+13b] showed how to generically transform an ABE scheme satisfying sub-linear decryption property into a scheme satisfying input-specific runtime property. We sketch the transformation below.

Suppose there exists an ABE scheme $\text{ABE}_{\text{sub}}$ that satisfies sub-linear decryption property. We construct $\text{ABE}^*$ as follows: execute $\log(T)$ copies of $\text{ABE}_{\text{sub}}$, where the maximum runtime bound in the $i^{th}$ copy of $\text{ABE}_i$ is set to be $2^i$. Note that $\text{ABE}^*$ is a secure ABE for RAMs scheme. In terms of public key, ciphertext and attribute key sizes in $\text{ABE}^*$ are larger than the corresponding parameters in $\text{ABE}_{\text{sub}}$ by a factor of $\log(T)$. However, the decryption time is input-specific: given a ciphertext of attribute $D$ and an attribute key of $P$, we use the $i^{th}$ copy of $\text{ABE}_{\text{sub}}$ where $i$ is the least integer such that the runtime of $P^D \leq 2^i$. This means that the decryption time grows with $2^i$ and not $T$.

Combining this with our construction satisfying sub-linear decryption complexity, we have:

**Theorem 5.12.** Assuming learning with errors, there exists an attribute-based encryption scheme for RAMs satisfying input-specific runtime property.

**References**


54


