Turing Machines with Shortcuts: Efficient Attribute-Based Encryption for Bounded Functions

Xavier Boyen and Qinyi Li(✉)

Queensland University of Technology, Brisbane, Australia
qinyi.li@student.qut.edu.au

Abstract. We propose a direct construction of attribute-based encryption (ABE) scheme for bounded multi-stack deterministic pushdown automata (DPDAs) and Turing machines that have polynomial runtime in the security parameter. Particularly, we show how to extend our construction to handle bounded DPDAs with two or more stacks, which leads to an ABE scheme for deterministic Turing machines (DTMs) with polynomial runtime.

Our ABE schemes have "input-specific" decryption runtime meaning that the decryption time depends on the semantics of attributes. If a machine halts prematurely on a certain input, its execution can be cut short. To the best of our knowledge, our ABE scheme is the first one that achieves this property and has security proofs based on standard cryptographic assumption.

The key technical ingredient we apply is a special graph encoding on the executions of bounded DPDAs with multi-stacks, allowing us to remember just enough of the execution history to enforce correct evaluation. The security of our scheme is shown to be based on the learning with errors (LWE) problem in the selective security model.

1 Introduction

Attribute-based encryption (ABE) enables the enforcement of complex access control conditions based on expressive decryption keys and ciphertext attributes. In a (key-policy) ABE scheme [15], a decryption key is associated with a Boolean predicate P from a family of predicates $\mathcal{P}$, and a message is encrypted with a public attribute string $\mathbf{w}$. Decryption succeeds iff $P(\mathbf{w}) = 1$. Designing ABE schemes with expressive and efficient access policies is of both theoretical and practical interest. On one hand, the research of ABE results in substantial advances in theoretical cryptography, such as functional encryption, garbling schemes and various of security proof techniques. With its plenty of applications in Cloud storage, access control, and outsourced computations etc., ABE has been shown a promising cryptographic primitive for the "Big Data" era in which huge amount of sensitive data needs to be securely stored and efficiently accessible in an expressive way.

M. Manulis et al. (Eds.): ACNS 2016, LNCS 9696, pp. 267–284, 2016.
DOI: 10.1007/978-3-319-39555-5_15

ABE with Input-Specific Decryption Complexity. Usually, the policy evaluation in decryption algorithms of ABE incurs a heavy computational overhead. As it is mentioned in [12], in most of the previously proposed ABE schemes whose polices are polynomial-size boolean formulas or boolean circuits, the policy evaluation always takes worst-case runtime. With the input-specific runtime, policy evaluation runs in the bare necessary time and could be very fast even for very large inputs. This property is crucial in most real world applications.

A simple example is the filter rule set in firewall systems. In a firewall system, the coming packets are inspected by filters according to a set of filtering rules. These filtering rules are usually sequentially arranged and will be applied to packets sequentially as well. Inspecting one packet under all rules is obviously inefficient. However, it is common practice to arrange the rules so that early accept/reject decisions can be made for most normal packets. As a consequence, decisions will hopefully be made very quickly for most of packets and, thus, efficiency increases. Our Turing machine-ABE scheme with input-specific runtime could see applications in complex filtering-based security systems for encrypted data.

From a practical perspective, it is desirable to design ABE schemes from standard cryptographic assumptions with efficient data access or decryption time. This serves for another motivation of us to design automata-based ABE scheme with input-specific decryption time. In the computation models of various automata, computations of an automaton would finish once it gets to an accept state after reading a prefix of inputs. However, this does not mean all the automata-based ABE schemes have input-specific decryption. In fact, the decryption time in the pairing-based ABE schemes in [2,26], depends on the length but not the content of input attributes, and thus, is not input-specific.

Most interestingly, it is a well-known (if poorly understood) phenomenon that most NP-complete problems exhibit a "phase transition" [9] from easy to hard to easy again, as a certain statistic of the input is varied (such as the ratio of number of clauses to number of variables in random 3-SAT instances). This is why SAT solvers work very well in practice, despite tackling problems that are formally NP-hard. Accordingly, our input-specific runtime constructions open the door to functional policy specifications that are technically NP-complete or NP-hard, but easily computable for inputs of practical interest.

Expressiveness of ABE. One of the central problems in ABE research is how to make predicates and policies ever more expressive. Most ABE schemes handle boolean formulas of polynomial size, with certain restrictions such as a monotonicity requirement. An advance over this model was provided with the pairing-based ABE schemes from [2,26] and lattice-based scheme from [6], where predicates attached to decryption keys are deterministic finite automata (DFAs), and attribute strings attached to ciphertexts are viewed as DFA input strings. Another recent breakthrough in that area came with the construction of ABE schemes for general boolean circuits with polynomial size [4,10,13], using different technical ideas such as multi-linear maps and/or lattices. Assuming the existence of Extractable Witness Encryption and Succinct Non-interactive

Arguments of Knowledge (SNARKS), two very strong and non-standard assumptions, Goldwasser et al. [12] proposed an ABE scheme for any polynomial-time Turing machines. An outstanding feature of the ABE schemes for DFAs from [2,26] and for Turing machines from [12] is that attribute strings could be arbitrarily long, which lead to various interesting applications, such as the innovative audit log system from [26]. However, it should be mentioned that the non-standard computational assumptions in [2,26] require fixing an a priori bound on the input length for the security reduction to obtain, and the construction of [12] came with a price of requirement of strong assumptions which have no satisfactory instantiations. In a different direction, ABE schemes for general circuits [4,10,13] provide obvious versatility benefits, as they enable predicates or policies to be any polynomial-size combinational (memoryless) function. For certain applications where policies have polynomial size, one could convert the policies into circuits on the fly and then apply the ABE schemes for circuits.

In this work, from the perspective of expressiveness of ABE, we focus on extending the notion of ABE for DFAs from [6,26] by providing *memory* taking the form of one or more push-down stacks—yielding a notion of ABE for (bounded) deterministic pushdown automata and Turing machines respectively.

Although the framework of [26] initiated the study of ABE schemes for automata (as opposed to the more traditional boolean formulas and circuits), it did not seem to support any mechanism for "read/write" tapes. Conceptually, two difficulties with implementing secure stack or tape machines are that configurations are exponentially many and time varying. One possible but not very satisfactory answer is to encode the entire memory (e.g., the stack or the tape) atomically into the atomic state or configuration of the machine. A related approach is to unroll an entire (bounded) Turing machine into a boolean circuit, e.g., so that a circuit-oriented ABE scheme [4,10,13] can be used; in this case it is the entire memory across the entire execution that is encoded "atomically" into the circuit as a function of its inputs.

A conceptually more desirable approach is to embrace the nature of the stack or tape as an attached memory, from which only the current element under the read/write head is accessible to the actual state machine, itself possibly very small. This approach requires guarantees that the portion of memory that is temporarily out of sight will not be tampered with—a non-trivial proposition. Our main contribution is to provide a secure way to attach stacks and tapes onto a "seed" ABE for DFA in a flexible and generic way. Based on this, we show how to realise ABE for deterministic state machines with multiple stacks (two stacks are enough to get a Turing machine, though additional stacks will increase efficiency). Our construction is based on the standard LWE assumption, rather than the pairing-based approach of [26].

1.1 Our Results

With above two motivations, we present a direct (key-policy) ABE scheme with input-specific decryption time for multi-stack deterministic pushdown automata

(DPDAs) that have polynomial (in security parameter) runtime. In particular, this yields a (key-policy) ABE scheme for deterministic Turing machines (DTMs) with polynomial (in security parameter) runtime through the equivalence between DTMs and DPDAs with two stacks. We refer to this special type of DPDAs and DTMs as bounded DPDAs and bounded DTMs, respectively, for short. To the best of our knowledge, our construction is the first ABE scheme which directly supports stack automata and has input-specific decryption time from standard cryptographic assumptions.

In our scheme, predicates or policies of users' decryption keys are directly expressed as bounded DPDAs. Messages are encrypted with polynomial-size attribute strings (the length of individual strings can vary). Decryption keys recover messages if and only if the automata recognize the strings attached to the ciphertexts. We prove the security of our scheme in the selective security model, based on the learning with errors problem (LWE).

In general, deterministic pushdown automata refer to single-stack machines. Deterministic pushdown automata with two stacks are much more powerful, being equivalent to deterministic Turing machines (DTMs). Our approach works with an arbitrary number of stacks; we will focus on constructing an ABE scheme for single-stack DPDAs and then show how to extend it easily to two (or more) stacks, to capture the full power of (bounded) deterministic Turing machines.

1.2 Our Approaches

Our approaches stem from the LWE-based ABE scheme for DFAs from [6]. We firstly give a quick review of stack and pushdown automata. A stack is a basic data structure which stores data in such a way that the most recently stored item is the first to be retrieved (also known as "last in, first out" access). Basically, stacks provide two principal operations: "push" a new element to the top of stack and "pop" the top-most element from stack. Stacks provide PDAs with additional memory space making PDAs more powerful than DFAs. A 3-Stack PDA is depicted in the Fig. 1.

We outline our approaches for 1-stack DPDAs. A deterministic pushdown automaton M with one stack is a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, s_0, z_0, F)$. The input

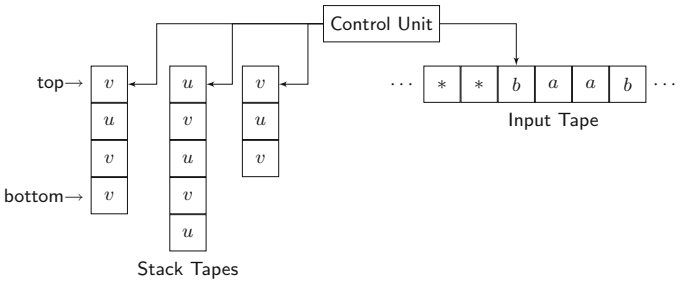

Fig. 1. A 3-Stack Pushdown Automaton

tape of M consists of symbols of an alphabet Σ. We write $\Sigma_\varepsilon = \Sigma \cup \varepsilon$ for the empty symbol ε. The finite internal states of M make up a set Q. $F \subseteq Q$ is a set of accept states. Γ is an alphabet of stack elements. For every execution, M starts from a unique initial state $s_0 \in Q$ and the stack is empty which is indicated by a bottom stack element $z_0 \in \Gamma$. For one computation process of M, a string $\mathbf{w}$ of symbols of Σ, which forms an input tape, is taken as an input. In one step of computation, M takes a current letter of input tape and the top stack element as input, does a stack operation and shifts its state according to a *deterministic* transition function $\delta : \Gamma \times Q \times \Sigma_\varepsilon \to \Gamma \times Q$. For instance, in the transition $\delta(u, s, b) \to (v, s')$, an automaton reads the symbol $b \in \Sigma$, changes the top stack element from u to v, and shifts its state from s to s'. Once M reaches some accept state $s_\omega \in F$, it stops and accepts $\mathbf{w}$. A PDA does not "remember" its execution history. A triple $(\boldsymbol{\gamma}, s, \mathbf{w})$, which is called Instantaneous Description (ID), can be used to track the execution history of a PDA by capturing a "snapshot" of a PDA's execution status. In the ID, $\boldsymbol{\gamma} \in \Gamma^*$ is the current stack contents with sequential order, $s \in Q$ is the current state, and $\mathbf{w} \in \Sigma^*$ is the remaining unread input. An accept ID is $(\boldsymbol{\gamma}', s_\omega, \mathbf{w}')$ where $\boldsymbol{\gamma} \in \Gamma^*$ is the stack contents, $s_\omega \in F$ is a accept state, and $\mathbf{w}' \in \Sigma^*$ is a suffix of $\mathbf{w}$.

The starting point of our construction is to naturally encode the input tapes and the transition function of PDAs into ciphertexts and decryption keys, respectively. We show the idea by describing a toy construction. Let ℓ be the upper bound of the length of all input strings. Let $\mathbf{S}_{z_0}$ and $\mathbf{A}_{s_0}$ be two publicly known matrices represent the unique initial stack element z_0 and state s_0 for all DPDAs. A lattice trapdoor of $[\mathbf{S}_{z_0}||\mathbf{A}_{s_0}]$ serves as the master secret key. A ciphertext of the input tape which contains an input string $\mathbf{w} = w_1 w_2 \dots w_\ell \in \Sigma^\ell$ has the LWE form

$$\mathbf{s}^\top [\mathbf{S}_{z_0}||\mathbf{A}_{s_0}||\mathbf{G}_{w_1}||\mathbf{G}_{w_2}||\dots||\mathbf{G}_{w_\ell}] + \boldsymbol{\nu}^\top$$

for a random secret vector $\mathbf{s} \xleftarrow{\$} \mathbb{Z}_q^n$, public matrices $\mathbf{G}_{w_i} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$, and noise vector $\boldsymbol{\nu}$ from some noise distribution χ. Note $[\mathbf{S}_{z_0}||\mathbf{A}_{s_0}||\mathbf{G}_{w_1}||\mathbf{G}_{w_2}||\dots||\mathbf{G}_{w_\ell}]$ is just the matrix representation of ID $(s_0, z_0, w_1 w_2 ... w_\ell)$ for all the DPDAs in the beginning of execution.

The decryption key of a PDA $M = (Q, \Sigma, \Gamma, \delta, s_0, z_0, F)$ contains a set of low-norm transition matrices $\{\mathbf{R}_\delta\}$ such that a transition $\delta(u, s, b) \to (v, s')$ (where $s \notin F$) is mathematically abstracted as a matrix multiplication $[\mathbf{S}_u||\mathbf{A}_s||\mathbf{G}_b]\mathbf{R}_\delta = [\mathbf{S}_v||\mathbf{A}_{s'}] \pmod q$ (here we don't consider the ε transitions). These equations are inductively constructed. Firstly check if matrix $[\mathbf{S}_u||\mathbf{A}_s]$ exists with trapdoor (the first of this type of matrix is $[\mathbf{S}_{z_0}||\mathbf{A}_{s_0}]$ which does have trapdoor). If one of the sub-matrices of $[\mathbf{S}_u||\mathbf{A}_s]$ does not exist, sample it by "matrix-trapdoor" sampling algorithm $\mathsf{TrapGen}$. Run $\mathsf{TrapGen}$ again to sample $\mathbf{S}_v$ and $\mathbf{A}_{s'}$ if they haven't been created. Then apply the lattice preimage sampling function $\mathsf{SampleD}$ to sample the low-norm transition matrix $\mathbf{R}_\delta$. Finally, each matrix $\mathbf{A}_{s_\omega}$ for $s_\omega \in F$ will be equipped with a decryption vector that allows decrypters to recover messages.

The decryption procedure works by following an accept execution path, sequentially applying the transition matrices $\mathbf{R}_\delta$ to linearly transform

$$\mathbf{s}^\top[\mathbf{S}_{z_0}||\mathbf{A}_{s_0}||\mathbf{G}_{w_1}||\mathbf{G}_{w_2}||\ldots||\mathbf{G}_{w_\ell}] + \boldsymbol{\nu}^\top$$

into

$$\mathbf{s}^\top[\mathbf{S}_u||\mathbf{A}_{s_\omega}||\mathbf{G}_{w_j}||\mathbf{G}_{w_{j+1}}||\ldots||\mathbf{G}_{w_\ell}] + \bar{\boldsymbol{\nu}}^\top,$$

(for some $u \in \Gamma$) which is the incomplete (in terms of stack configuration) matrix representation of an accept ID $(s_\omega, \boldsymbol{\gamma}, w_j w_{j+1} ... w_\ell)$ where $1 \leq j$. The decryption vector of $\mathbf{A}_{s_\omega}$ then applies. We note the decryption does not need to read the whole input string.

Two problems make this toy construction insecure. Firstly, the concatenation matrix $[\mathbf{G}_{w_1}||\mathbf{G}_{w_2}||\ldots||\mathbf{G}_{w_\ell}]$ does not mathematically enforce any sequence between letters in the string. Secondly, no mechanism ensures the consistency of stack configurations. Different configurations may lead to the same execution result. For instance, consider two IDs $(uvz_0, s, w_1w_2w_3)$ and $(uvvz_0, s, w_1w_2w_3)$. A transition $\delta(u, s, w_1) \to (v, s')$ takes them to different next IDs, but they have the same transition equation. In particular, a transition equation only mirrors one piece of the whole execution of a DPDA and shows no connections with other transitions. On the other hand, this problem brought by "memoryless" description of DPDAs can be amended by including the whole execution history to each transition step, just as ID does. However, exponentially many IDs (within the input length bound ℓ) is impossible to be encoded in decryption keys.

Summing up, in order to securely embed bounded DPDAs with one or multiple stacks into decryption keys, we must take care to retain enough state to prevent malleability attacks while avoiding an exponential blow-up of the size of the decryption keys.

The Challenges of (Not) Keeping Memory. To figure out the minimal amount of state we need to keep in a pushdown automaton with respect to some input length bound, we notionally unroll all the possible execution paths of the automaton into a specially crafted low-dimensional space. The dimensions and coordinates of the space are determined so that the directed execution graph is *acyclic* (no directed cycles). Intuitively, the coordinates of this "execution graph space" represent the variables that the core state machine must remember about the stack/tape configuration. For example, one dimension could be a counter indicating how many symbols the machine has read from its second stack so far. The fewer the dimensions, the more risk that the graph of all possible unrolled executions, will contain directed cycles. If that happens, a malicious user can "jump" from one execution to another, yielding an illegal execution that could accept a word not in the language—an attack.

At one extreme, a machine whose core remembers nothing about the tapes corresponds to a very low-dimensional execution space (as the case of the toy construction). In this case, an adversary will be able to alter the stack tape or the input tape "out of sight" of the state machine, undetected, in order to take shortcuts/longcuts/sidecuts in the execution, throwing the machine into

a configuration it should not have been able to reach from the given input. Conversely, a Turing machine whose state remembers everything, including the whole tape itself, can obviously be simulated as a mere DFA, albeit one with a huge state space, without danger of allowing an adversary to deviate from the execution. In this case the execution graph space will have (at least) one dimension for each possible cell. This will make the graph acyclic, but at the cost of requiring a state machine with exponentially many states.

Between those unworkable extremes, we devise an execution graph space of low constant dimension (function only of the *number* of stacks) that will always guarantee acyclicity. This gives us the variables that the core state machine will need to keep track of, to cryptographically ensure correct execution regardless of the size of the stack(s). Those variables enforce a set of constraints sufficient to ensure that users (performing the decryption) consistently proceed with the forward execution of the DPDA step by step, without having to remember unneeded data about the DPDA's previous steps. In our full construction, the actual execution of an automaton is based on the repeated application of transition equations similar to the toy construction. But each transition equation will carry (just) enough aforementioned variables so that the acyclic execution graphs can be correctly instantiated. We note that the transition equation (as in the toy construction) is a direct generalisaztion of the two-to-one recoding [13], itself ultimately based on lattice basis delegation [1,8].

Perhaps the most novel aspect of our ABE construction is the secure construction of (long, bidirectional) read/write tapes that do not require the entire configuration to be kept "in focus" at all times, unlike most other lattice-based public-key encryption scheme. We cryptographically enforce "big picture" consistency across space and time, using only local transformations or transitions with a short window of visibility. More specifically, there are a few transition steps involved in our constructions, implemented as *local* matrix multiplications. Once we ensure the current decryption step has been securely and honestly taken, it is automatically guaranteed that the previous decryption steps are securely taken, which by induction implies that the entire execution is *globally* correct. We prove all this using a game-based reductionist simulation from the LWE hardness assumption. The aforementioned execution graph and the selection of its dimensions to ensure acyclicity, are critical to the success of this simulation.

Relationship with ABE for Circuits. We emphasize that the pushdown automata in our scheme are subject to polynomial size input and polynomial execution time restrictions, and thus also admit (non unique) functionally equivalent polynomial size circuits. ABE schemes for general circuits like [4,10,13] can thus in theory achieve the same functionality by converting the bounded DPDA to DTM into an equivalent circuit beforehand. Our schemes provide a quite different and direct way to solve the problem, with an additional advantage of input-specific decryption time.

More specifically, the process of converting deterministic pushdown automata (especially with multiple stacks) into circuits is subtle. First, we need an actual algorithm, not just an existential equivalence. Second, the actually translated

circuits can be optimised to have gates with small fan-in at the expense of depth, or be optimised for shallowness and require many gates, but not both. As both the height of the circuit and the number of gates and their fan-in affect the efficiency of circuit-based ABE schemes, a compromise would have to be struck.

A proper comparison with circuit-based ABE would require taking into account the exact cost of translating a DPDA or Turing machine into a circuit, not only in the sense of existential upper bounds, but also in the form of efficient algorithms that achieve them. A naive translation approach could result in a very noticeable penalty in the resulting Circuit ABE. Alas, an extensive literature search has failed to reveal any hypothetical TM-to-Circuit translators that would be markedly superior than the naive translation. For an ABE policy specified as a DPDA or a DTM, our construction sidesteps all issues related to translation and its tuning, and has the advantage of simplicity. Conversely of course, given an ABE policy as a circuit, it would be preferable to use a DTM for circuits rather translate it into a machine representation for our construction. Last but not least, the circuits conversion will certainly lose the advantage of input-specific decryption times from which many potential applications may get benefits in terms of efficiency.

The main contribution of our result is to show how the "simple" idea of *directly* embedding a bounded DPDA or DTM into an ABE system can actually be made to work, and proven secure in the reductionist simulation framework.

1.3 Other Related Works

The research on ABE can be traced back to the development of identity-based encryption (IBE). fuzzy IBE, a variant of IBE introduced in [23], triggered the birth of ABE. Various ABE schemes have been proposed based on multi-linear maps, bilinear maps and lattices [5,15,18,25]. While ABE enables complex access mechanism on encrypted data, it only provides privacy for the payload messages rather than attributes of the message. ABE is inadequate in some applications where the attributes themselves are considered to be sensitive. Predicate encryption (PE) has a similar structure to ABE but enjoys stronger privacy. PE hides the attributes of encrypted data as well for decryptors whose predicates of decryption are not satisfied by the attributes. Very recently, Gorbunov et al. [14] proposed a PE scheme for general circuits from standard LWE assumption by combining the key-homomorphic ABE scheme for circuits [4] and LWE-based fully homomorphic encryption. Functional encryption is a more powerful primitive which generalises ABE and PE. However, practical functional encryption schemes are only known for limited functionalities such as inner-product [17,20].

We note that the effort of improving (worst-case) decryption time of ABE schemes has been taken, for instance, in [16]. However, we are not aware of any existing ABE schemes from standard cryptographic assumptions that have the input-specific decryption time (the construction from [12] relies on SNARKS and extractable witness encryption, two very strong assumptions).

2 Preliminary

2.1 Lattices

We use the standard definitions of random integer lattices, discrete Gaussian distribution, the well-known lattice trapdoor techniques, and discrete Gaussian sampling algorithems. Particularly, we use the trapdoor generation algorithm $\mathsf{TrapGen}$ from [1,11,19] in the black-box way. The sampling algorithms $\mathsf{SampleD}$ and $\mathsf{SampleExtend}$ used in this paper are respectively the same as the $\mathsf{SamplePre}$ and $\mathsf{SampLeft}$ algorithms defined in [1]. We refer to the full version of this paper or above citations for details. The security of our constructions is based on the learning with errors (LWE) problem firstly introduced by Regev [22]. We refer to [7,21,22] for the definition and hardness results of the LWE problem.

2.2 Pushdown Automata

A *deterministic* **pushdown automaton** with one stack is a 7-tuple $(Q, \Sigma, \Gamma, \delta, s_0, z_0, F)$. Q is a finite set of states. $F \subseteq Q$ is the set of the accept states. Σ is the finite input alphabet. $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ for the empty symbol ε. Γ is the finite stack alphabet. $\delta : \Gamma \times Q \setminus F \times \Sigma_\varepsilon \to \Gamma \times Q$ is the deterministic transition function. $z_0 \in \Gamma$ is the initial stack element and it is always at the bottom of the stack and never been removed. $s_0 \in Q$ is the unique start state. The reader is referred to the full version of this paper or the textbook [24] for the computational models of deterministic pushdown automata and Turing machines.

2.3 Definitions of Attribute-Based Encryption for PDAs

A key-policy attribute-based encryption scheme for PDAs consists of four algorithms (Setup, KeyGen, $\mathsf{Encrypt}$, $\mathsf{Decrypt}$). Setup takes as input a security parameter λ and a universal alphabet Σ_ε. It generates public parameters Pub and master secret key Msk. KeyGen uses Msk to generate decryption Sk_M for a given PDA machine M. $\mathsf{Encrypt}$ applies Pub to encrypt a message Msg under a string $\mathbf{w} \in \Sigma^*$, and produces the ciphertext $\mathsf{Ctx}_{\mathbf{w}}$. $\mathsf{Decrypt}$ recovers the message from $\mathsf{Ctx}_{\mathbf{w}}$ using Sk_M if M accepts $\mathbf{w}$, i.e. $\mathbf{w} \in L(M)$.

Security Model. We review the game-based selective security definition of (key-policy) ABE scheme for PDAs. Let $\mathcal{A}$ be the adversary, $\mathcal{B}$ be the challenger.

Initial. $\mathcal{A}$ submits a string $\mathbf{w}^* \in \Sigma^*$ as its challenge.

Setup. $\mathcal{B}$ runs algorithm Setup to generate the public parameters Pub and master secret key Msk and passes Pub to $\mathcal{A}$.

Phase 1. $\mathcal{A}$ adaptively issues the key generation queries for keys correspond to any PDA machine M of its choice. The only restriction is $\mathbf{w}^* \notin L(M)$. $\mathcal{B}$ runs the algorithm $\mathsf{KeyGen}(\mathsf{Pub}, \mathsf{Msk}, M)$ and returns Sk_M.

Challenge. $\mathcal{A}$ chooses a challenge message to be encrypted with $\mathbf{w}^*$. $\mathcal{B}$ flips a random coin $\gamma \in \{0, 1\}$. If $\gamma = 1$, the challenge ciphertext is returned. Otherwise, a random element in the ciphertext space is returned.

Phase 2. This phase is exactly the same as **Phase 1**.
Guess. Finally, $\mathcal{A}$ outputs a guess bit γ' of γ. It wins if $\gamma' = \gamma$.

The advantage of $\mathcal{A}$ in the above game is defined as $|\Pr[\gamma' = \gamma] - \frac{1}{2}|$. We say a (Key-Policy) ABE scheme for PDAs is selectively secure if all PPT adversaries have at most a negligible advantage in the above game. In the stronger model named the adaptive security model, adversary submits the challenge string $\mathbf{w}^*$ in the **Challenge** phase.

3 Execution Graph of DPDAs

We now turn to explain the structure of the specially crafted execution graphs of DPDAs which are low-dimensional and acyclic. The execution graphs allow us to securely encode bounded DPDAs into decryption keys and to successfully overcome the difficulties of keeping execution history in memory. Without loss of generality, 1-Stack DPDAs are described in full detail. It is straightforward to extend the ideas to 2-Stack DPDAs and n-Stack DPDAs, by linearly increasing the dimension of the execution graphs so that they remain acyclic.

3.1 Descriptions of Execution Graph

The execution graph $G = (V, E)$ of a 1-Stack DPDA $M = (Q, \Sigma, \Gamma, \delta, s_0, z_0, F)$, with respect to the input length bound τ and running time bound η, consists of a set of vertices, denoted by V, and a set of edges denoted by E.

A vertex in V, which comprises of 5 variables, has form (u, s, j, t', t). The first coordinate $u \in \Gamma$ is the current top-most stack element. The second coordinate $s \in Q$ is the current state of DPDA. The third coordinate j, where $1 \leq j \leq \tau$, is the "input position" indicating currently the first $j - 1$ input symbols have been read by M, and the next symbol to be read is the j^{th} one. The fourth and fifth coordinate t' and t, where $-1 \leq t' \leq 2\eta - 1$, $0 \leq t \leq 2\eta$ and $t' < t$, are "stack position tags" of u. t represents the stack position of current stack element u and t' represents the stack position of u's previous stack element. These stack position tags sequentially chain all current stack elements together in a logical way. During the transition (execution), the stack position tags change dynamically (in a way we specify later) so that the new top-stack element which is pushed in has a (logically) higher position than the previous top stack element, and the sequential relation between the stack element that is popped out, and rest of stack elements is removed. For the special stack element z_0 which will never be popped out, we assign the special tag value -1 to indicate that z_0 is always in the bottom of the stack.

The edges in E are defined by the input of transitions, either a symbol $b \in \Sigma$ or an empty symbol ε. The input symbol $b \in \Sigma$, which defines the outgoing edge of a vertex (s, u, j, t', t), is tied by input position j. We don't explicitly defines the input position for the ε input as it never appears in the input string.

The initial vertex of an execution graph of M is $(z_0, s_0, 1, -1, 0)$. Inductively, two vertices are connected to each other with respect to the type of transition, increasing of input position tags and increasing of stack positions. Let (u, s, j, t', t) be the current vertex for the top stack element $u \in \Gamma$, the state $s \in Q$, input position j and stack position tags t' and t for u and u's previous stack element respectively.

1. For a "push" transition $\delta(u, s, b) \to (v, s')$, $b \in \Sigma$ will define the outgoing edge. The next vertex will be defined as $(v, s', j+1, t+1, t+2)$ in which we increase the input position by 1, meaning that a non-empty symbol is read, and assign stack position tag $t+2$ to v and update u's position tag from t to $t+1$. There must be the case that $-1 \leq t' < t$. We write this relation as:

$$(u, s, j, t', t) \xrightarrow{b} (v, s', j+1, t+1, t+2)$$

   In the special case $t' = -1$, we have $u = z_0$.
2. For a "pop" transition $\delta(s, u, b) \to (s', v)$, $b \in \Sigma$ will define the outgoing edge. The next vertex will be defined as $(v, s', j+1, t'', t+1)$ in which we increase the input position by 1 and update v's stack position tag from t' to $t+1$ and v's previous tack element's stack position tag, say t'', is not change. There must be the case that $-1 \leq t'' < t' < t$. We write this relation as:

$$(u, s, j, t', t) \xrightarrow{b} (v, s', j+1, t'', t+1)$$

   In the special case $t'' = -1$, we have $v = z_0$.
3. For a ε "push" transition $\delta(u, s, \varepsilon) \to (v, s')$, ε symbol will define the outgoing edge. The next vertex will be defined as $(v, s', j, t+1, t+2)$ in which the input position stays unchanged, meaning that no input has been read in this transition. v is assigned the new stack position tag $t+2$ and u's position tag is updated from t to $t+1$. There must be the case that $-1 \leq t' < t < 2\eta$. We write this relation as:

$$(u, s, j, t', t) \xrightarrow{\varepsilon} (v, s', j, t+1, t+2)$$

   In the special case $t' = -1$, we have $u = z_0$.
4. For a ε "pop" transition $\delta(u, s, \varepsilon) \to (v, s')$, ε symbol will define the outgoing edge. The next vertex will be defined as $(s', v, j, t'', t+1)$ in which the input position stays unchanged, meaning that no input has been read in this transition. We update v's stack position tag from t' to $t+1$. v's previous tack element's stack position tag, say t'', is not change. There must be the case that $-1 \leq t'' < t' < t < 2\eta$. We write this relation as:

$$(u, s, j, t', t) \xrightarrow{\varepsilon} (s', v, j, t'', t+1)$$

   In the special case $t'' = -1$, we have $v = z_0$.

In a execution process with respect to a specific input, M will start from $(z_0, s_0, 1, -1, 0)$. It then follows the path defined by the input and ε transitions

to travel between vertices. Once M reaches a vertex with some accept state s_ω, it stops and accepts the input. Otherwise, M stops and rejects the input if there is no transition with respect the current input or j, t', t reach the bounds.

To see why the execution graphs for 1-Stack DPDAs are acyclic, the coordinates of tuple (u, s, j, t', t), (input position j and stack position tags t', t) increase monotonically with at least one of them increasing at each step. t' decreases only when the top-most stack elements are popped out from the stack and its stack position tag is never going to be used again.

3.2 Matrix Representation

In our constructions, the execution graphs are instantiated by Matrices (recall the toy construction). However, matrix concatenation neither forces any sequential order to the individual matrices nor logically binds the individual matrices together. On the other hand, in execution graph, coordinates in a vertex are logically integrated, vertices and edges (specified by input symbols) are bound with respect to the input positions. In order to mitigate this problem, we use subscripts of matrices to denote the state, stack element and input symbols, and encode the input positions and stack position tags in the superscripts of matrices to tie concatenated matrices together logically.

Specifically, for a vertex (u, s, j, t', t), we encode it by matrix concatenation $[\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}]$. $\mathbf{S}_u^{(t',t)} \in \mathbb{Z}_q^{n\times m}$ is the stack matrix of u with t as u's stack position tag and t' as the stack position tag of u's previous element. $\mathbf{A}_s^{(t,j)} \in \mathbb{Z}_q^{n\times m}$ is the state matrix of s. The superscript t ties $\mathbf{A}_s^{(t,j)}$ to $\mathbf{S}_u^{(t',t)}$, and j ties $\mathbf{A}_s^{(t,j)}$ to the j^{th} input symbol, say b, which has matrix representation $\mathbf{G}_b^{(j)}$.

For expressing the transition equations that connect two vertices through an edge, we consider the following cases:

- $(u, s, j, t', t) \xrightarrow{b} (v, s', j+1, t+1, t+2)$ where the push transition $\delta(u, s, b) \to (v, s')$ for j^{th} input b happens: sample a low-norm transition matrix $\mathbf{R}_\delta^{(t',t,j)} \in \mathbb{Z}^{3m\times 3m}$ such that:

$$[\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}||\mathbf{G}_b^{(j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_u^{(t',t+1)}||\mathbf{S}_v^{(t+1,t+2)}||\mathbf{A}_{s'}^{(t+2,j+1)}] \pmod q$$

- $(u, s, j, t', t) \xrightarrow{b} (v, s', j+1, t'', t+1)$ where the pop transition $\delta(u, s, b) \to (v, s')$ for j^{th} input b happens: sample a low-norm transition matrix $\mathbf{R}_\delta^{(t',t,j)} \in \mathbb{Z}^{4m\times 2m}$ such that:

$$[\mathbf{S}_v^{(t'',t')}||\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}||\mathbf{G}_b^{(j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_v^{(t'',t+1)}||\mathbf{A}_{s'}^{(t+1,j+1)}] \pmod q$$

- $(u, s, j, t', t) \xrightarrow{\varepsilon} (v, s', j, t+1, t+2)$ where the push transition $\delta(u, s, \varepsilon) \to (v, s')$ for ε input happens: sample a low-norm transition matrix $\mathbf{R}_\delta^{(t',t,j)} \in \mathbb{Z}^{2m\times 3m}$ such that:

$$[\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_u^{(t',t+1)}||\mathbf{S}_v^{(t+1,t+2)}||\mathbf{A}_{s'}^{(t+2,j)}] \pmod q$$

– $(u, s, j, t', t) \xrightarrow{\varepsilon} (s', v, j, t'', t+1)$ where the push transition $\delta(u, s, \varepsilon) \to (v, s')$ for ε input happens: sample a low-norm transition matrix $\mathbf{R}_{\delta}^{(t',t,j)} \in \mathbb{Z}^{3m \times 2m}$ such that:

$$[\mathbf{S}_v^{(t'',t')} || \mathbf{S}_u^{(t',t)} || \mathbf{A}_s^{(t,j)}] \mathbf{R}_{\delta}^{(t',t,j)} = [\mathbf{S}_v^{(t'',t+1)} || \mathbf{A}_{s'}^{(t+1,j)}] \pmod q$$

In all above cases, if the state s' in the target vertex equals to some accept state s_ω, we will simply use matrix $\mathbf{A}_\omega$ to denote it without any superscripts. This is because the vertex (u, s_ω, j, t', t) must be the terminal of the execution path and no next vertex exists.

The reason we use slightly different forms in the equations for push and pop transition is that we update the stack position tags of the stack elements to keep the actual transition equations compatible with the execution graphs such that no direct cycles happen.

4 The ABE Scheme for Bounded 1-Stack DPDAs

4.1 Construction

$\mathsf{Setup}\big(1^\lambda, 1^\tau, 1^\eta, \Sigma_\varepsilon = \{0, 1, \varepsilon\}\big)$ On input of security parameter 1^λ, upper bound $\tau = \tau(\lambda)$ of length of input string, upper bound $\eta = \eta(\lambda)$ of running time of the pushdown automata, and the universal alphabet Σ_ε:

1. Pick 2τ matrices $\mathbf{G}_b^{(j)} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$ for $j \in [\tau]$ and $b \in \{0, 1\}$.
2. Sample $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and its trapdoor $\mathbf{T}_\mathbf{A} \in \mathbb{Z}^{m \times m}$ by $\mathsf{TrapGen}$.
3. Pick $\mathbf{u} \xleftarrow{\$} \mathbb{Z}_q^n$.
4. Output $\mathsf{Pub} = \left(\{\mathbf{G}_b^{(j)}\}_{b \in \{0,1\}, j \in [\tau]}, \mathbf{A}, \mathbf{u}\right)$ and $\mathsf{Msk} = \mathbf{T}_\mathbf{A}$.

$\mathsf{KeyGen}(\mathsf{Pub}, \mathsf{Msk}, M = (Q, \Gamma, \delta, s_0, z_0, F))]$ On input of Pub, Msk and a 1-Stack DPDA M, unroll M (up to the fixed bounds) into an execution graph:

1. Prepare for the initial vertex $(z_0, s_0, 1, -1, 0)$:
   (a) Run $\mathsf{TrapGen}$ to sample matrix $\mathbf{A}_{s_0}^{(0,1)}$ and its trapdoor.
   (b) Pick $\mathbf{S}_{z_0}^{(-1,0)} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$.
   (c) Sample $\mathbf{R} \in \mathbb{Z}^{m \times 2m}$ by $\mathsf{SampleD}$ such that $\mathbf{AR} = [\mathbf{S}_{z_0}^{(-1,0)} || \mathbf{A}_{s_0}^{(0,1)}] \pmod q$.
2. For a normal "push" transition $\delta(u, s, b) \to (v, s')$ that connects two vertices (u, s, j, t', t) and $(v, s', j+1, t+1, t+2)$ through the edge b at input position j, the algorithm does:
   (a) The state matrix $\mathbf{A}_s^{(t,j)}$ with trapdoor and stack matrix $\mathbf{S}_u^{(t',t)}$ are already defined.
   (b) In case $s' \in Q \setminus F$, run $\mathsf{TrapGen}$ to sample matrix $\mathbf{A}_{s'}^{(t+2,j+1)}$ and its trapdoor if such state matrix has not been defined. If s' equals to some accept state $s_\omega \in F$, and the matrix $\mathbf{A}_{s_\omega}$ has not been defined, run $\mathsf{TrapGen}$ to sample it with trapdoor.
   (c) Pick $\mathbf{S}_u^{(t',t+1)}, \mathbf{S}_v^{(t+1,t+2)} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$ for $v \in \Gamma$ if this stack matrix has not been defined.

(d) Run $\mathsf{SampleExtend}$ to sample $\mathbf{R}_\delta^{(t',t,j)}$ with distribution $(D_{\mathbb{Z}^{3m},\sigma})^{3m}$ such that if $s' \in Q \setminus F$:

$$[\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}||\mathbf{G}_b^{(j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_u^{(t',t+1)}||\mathbf{S}_v^{(t+1,t+2)}||\mathbf{A}_{s'}^{(t+2,j+1)}] \pmod q$$

or $s' = s_\omega$ for some $s_\omega \in F$:

$$[\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}||\mathbf{G}_b^{(j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_u^{(t',t+1)}||\mathbf{S}_v^{(t+1,t+2)}||\mathbf{A}_{s_\omega}] \pmod q.$$

3. For a normal "pop" transition $\delta(u,s,b) \to (v,s')$ that connects two vertices (u,s,j,t',t) and $(v,s',j+1,t'',t+1)$ through the input b at input position j, the algorithm does:
   (a) The state matrix $\mathbf{A}_s^{(t,j)}$ with trapdoor, stack matrices $\mathbf{S}_u^{(t',t)}$, $\mathbf{S}_v^{(t'',t')}$ are already defined.
   (b) In case that $s' \notin F$, run $\mathsf{TrapGen}$ to sample matrix $\mathbf{A}_{s'}^{(t+1,j+1)}$ and its trapdoor if such state matrix has not been defined. If s' is some accept state $s_\omega \in F$, and the matrix $\mathbf{A}_{s_\omega}$ has not been defined, run $\mathsf{TrapGen}$ to sample it with trapdoor.
   (c) Pick $\mathbf{S}_v^{(t'',t+1)} \xleftarrow{\$} \mathbb{Z}_q^{n\times m}$ if this stack matrix has not been defined.
   (d) Run $\mathsf{SampleExtend}$ to sample $\mathbf{R}_\delta^{(t',t,j)}$ with distribution $(D_{\mathbb{Z}^{4m},\sigma})^{2m}$ such that if $s' \notin F$:

$$[\mathbf{S}_v^{(t'',t')}||\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}||\mathbf{G}_b^{(j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_v^{(t'',t+1)}||\mathbf{A}_{s'}^{(t+1,j+1)}] \pmod q$$

   or $s' = s_\omega$ for some $s_\omega \in F$:

$$[\mathbf{S}_v^{(t'',t')}||\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}||\mathbf{G}_b^{(j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_v^{(t'',t+1)}||\mathbf{A}_{s_\omega}] \pmod q.$$

4. For a ε "push" transition $\delta(u,s,\varepsilon) \to (v,s')$ that connects two vertices (u,s,j,t',t) and $(v,s',j,t+1,t+2)$ through edge ε, the algorithm does:
   (a) The state matrix $\mathbf{A}_s^{(t,j)}$ with trapdoor and stack matrix $\mathbf{S}_u^{(t',t)}$ are already defined.
   (b) Run $\mathsf{TrapGen}$ to sample matrix $\mathbf{A}_{s'}^{(t+2,j)}$ and its trapdoor if $s' \notin F$ and such state matrix has not been defined. If s' is some accept state $s_\omega \in F$, and the matrix $\mathbf{A}_{s_\omega}$ has not been defined, run $\mathsf{TrapGen}$ to sample it with trapdoor.
   (c) Pick $\mathbf{S}_u^{(t',t+1)}, \mathbf{S}_v^{(t+1,t+2)} \xleftarrow{\$} \mathbb{Z}_q^{n\times m}$ if they haven't been defined.
   (d) Run $\mathsf{SampleExtend}$ to sample $\mathbf{R}_\delta^{(t',t,j)}$ with distribution $(D_{\mathbb{Z}^{2m},\sigma})^{3m}$ such that if $s' \notin F$:

$$[\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_u^{(t',t+1)}||\mathbf{S}_v^{(t+1,t+2)}||\mathbf{A}_{s'}^{(t+2,j)}] \pmod q$$

   or $s' = s_\omega$ for some $s_\omega \in F$:

$$[\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_u^{(t',t+1)}||\mathbf{S}_v^{(t+1,t+2)}||\mathbf{A}_{s_\omega}] \pmod q.$$

5. For a ε "pop" transition $\delta(u, s, b) \to (v, s')$ that connects two vertices (u, s, j, t', t) and $(v, s', j, t'', t+1)$ through the edge ε, the algorithm does:
   (a) The state matrix $\mathbf{A}_s^{(t,j)}$ with trapdoor, stack matrices $\mathbf{S}_u^{(t',t)}$, $\mathbf{S}_v^{(t'',t')}$ are already defined.
   (b) Run $\mathsf{TrapGen}$ to sample matrix $\mathbf{A}_{s'}^{(t+1,j)}$ and its trapdoor if $s' \notin F$ and such state matrix has not been defined. If s' is some accept state $s_\omega \in F$, and the matrix $\mathbf{A}_{s_\omega}$ has not been defined, run $\mathsf{TrapGen}$ to sample it with trapdoor.
   (c) Pick $\mathbf{S}_v^{(t'',t+1)} \xleftarrow{\$} \mathbb{Z}_q^{n\times m}$ if this stack matrix has not been defined.
   (d) Run $\mathsf{SampleExtend}$ to sample $\mathbf{R}_\delta^{(t',t,j)}$ with distribution $(D_{\mathbb{Z}^{3m},\sigma})^{2m}$ such that if $s' \notin F$:

$$[\mathbf{S}_v^{(t'',t')}||\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_v^{(t'',t+1)}||\mathbf{A}_{s'}^{(t+1,j)}] \pmod q$$

   or $s' = s_\omega$ for some $s_\omega \in F$:

$$[\mathbf{S}_v^{(t'',t')}||\mathbf{S}_u^{(t',t)}||\mathbf{A}_s^{(t,j)}]\mathbf{R}_\delta^{(t',t,j)} = [\mathbf{S}_v^{(t'',t+1)}||\mathbf{A}_{s_\omega}] \pmod q.$$

6. For all state matrices $\mathbf{A}_\omega$ of accept states $s_\omega \in F$, run $\mathsf{SampleD}$ to sample Gaussian vector $\mathbf{d}_{s_\omega}$ such that: $\mathbf{A}_{s_\omega}\mathbf{d}_{s_\omega} = \mathbf{u} \pmod q$.
7. Output the decryption key as:

$$\mathsf{Sk}_M = \left(\mathbf{R}, \{\mathbf{R}_\delta^{(t',t,j)}\}, \{\mathbf{d}_{s_\omega}\}_{s_\omega\in F}\right)$$

$\mathsf{Encrypt}(\mathsf{Pub}, \mathbf{w}, \mathsf{Msg})$ The encryption algorithm takes as input the public parameters Pub, a binary string $\mathbf{w}$ with length $\ell \le \tau$, and a message bit $\mathsf{Msg} \in \{0, 1\}$. Denote the i^{th} bit of $\mathbf{w}$ by $\mathbf{w}[i]$. The algorithm then does:

1. Randomly select a vector $\mathbf{s} \xleftarrow{\$} \mathbb{Z}_q^n$.
2. Select a noise scalar $\nu_0 \leftarrow \chi$ and compute the scalar

$$c_0 = \mathbf{s}^\top \mathbf{u} + \nu_0 + \mathsf{Msg}\lfloor q/2 \rfloor.$$

3. Select a noise vector $\boldsymbol{\nu}_1 \leftarrow \chi^{(\ell+1)m}$, and compute the vector

$$\mathbf{c}_1^\top = \mathbf{s}^\top[\ \mathbf{A}||\mathbf{G}_{\mathbf{w}[1]}^{(1)}\ ||\ \mathbf{G}_{\mathbf{w}[2]}^{(2)}\ ||\ \cdots\ ||\ \mathbf{G}_{\mathbf{w}[\ell-1]}^{(\ell-1)}\ ||\ \mathbf{G}_{\mathbf{w}[\ell]}^{(\ell)}\ ] + \boldsymbol{\nu}_1^\top.$$

4. Output the ciphertext for the attribute input string $\mathbf{w}$ as

$$\mathsf{Ctx}_\mathbf{w} = (c_0, \mathbf{c}_1).$$

$\mathsf{Decrypt}(\mathsf{Pub}, \mathsf{Sk}_M, \mathsf{Ctx}_\mathbf{w}, \mathbf{w})$ On input Pub, decryption key Sk_M of automaton M, ℓ-length attribute string $\mathbf{w} = w_1 w_2 \ldots w_\ell$ and the ciphertext $\mathsf{Ctx}_\mathbf{w}$ encrypted by $\mathbf{w}$.

1. If $\mathbf{w} \notin L(M)$, return an error symbol $\perp$. Otherwise, unroll the execution graph of M, find the execution path from the start state s_0 to an accept state s_ω. Assume M digests the first $\ell' \le \ell$ input symbols to get to s_ω. Collect all the transition matrices $\{\mathbf{R}_\delta^{t',t,j}\}$ (including $\mathbf{R}$) of the path and the vector $\mathbf{d}_{s_\omega}$.

2. Get the useful part of $\mathbf{c}_1$: $\bar{\mathbf{c}}_1^\top = \mathbf{s}^\top [\ \mathbf{A}\ ||\ \mathbf{G}^{(1)}_{\mathbf{w}[1]}\ ||\cdots||\ \mathbf{G}^{(\ell')}_{\mathbf{w}[\ell']}\ ] + \bar{\boldsymbol{\nu}}_1^\top$.
3. Set $\mathbf{c}_{1,0}^\top = \bar{\mathbf{c}}_1^\top \underbrace{\begin{bmatrix} \mathbf{R} & & & \\ & \mathbf{I}_m & & \\ & & \ddots & \\ & & & \mathbf{I}_m \end{bmatrix}}_{\in \mathbb{Z}^{(\ell'+1)m \times (\ell'+2)m}} = \mathbf{s}^\top [\mathbf{S}^{(-1,0)}_{z_0} || \mathbf{A}^{(0,1)}_{s_0} || \mathbf{G}^{(1)}_{\mathbf{w}[1]} || \cdots || \mathbf{G}^{(\ell')}_{\mathbf{w}[\ell']}] + \bar{\boldsymbol{\nu}}_{1,0}^\top$.
4. Sequentially apply the transition matrices to transfer $\mathbf{c}_{1,0}$ to get $\mathbf{c}_{1,end}^\top = \mathbf{s}^\top \mathbf{A}_{s_\omega} + \boldsymbol{\nu}_{1,end}^\top$. This can be done in an obvious way and the ciphertext part of stack matrices that come out with $\mathbf{A}_{s_\omega}$ at the last step is simply discarded.
5. Set $\Delta = c_0 - \mathbf{c}_{1,end}^\top \mathbf{d}_{s_\omega}$ and output output $\mathsf{Msg} = 0$ if $\|\Delta\| < q/4$, or $\mathsf{Msg} = 1$ otherwise.

4.2 Correctness and Parameters

We refer to the full version of this paper for the correctness and parameters selection of above scheme.

4.3 Security

Theorem 1. *The scheme is selectively secure if the $LWE_{n,q,\chi}$ problem is hard.*

We refer to the full version of this paper for the full proof. We also remark that as we consider attribute strings with bounded length, by relying on the sub-exponential hardness of the LWE problem and the standard "complexity leveraging" argument [3], above scheme is also adaptively secure.

5 Extensions to 2-Stack DPDAs (and Thus DTMs)

To extend the ABE scheme for 1-Stack DPDAs to handle two or more stacks (thus Turing machines), the execution graphs are correspondingly extended to represent additional memory and preserve acyclic property. This can be achieved by adding enough (linearly many) states. The resulting execution graphs have dimensions which are linearly more than the execution graphs of 1-Stack DPDAs in the number of stacks. The ABE schemes for multi-stacks DPDAs (Turing machines) are obtained by incorporating new execution graphs into matrix transition equations. Their security can be proved in a similar way of security proof of the ABE scheme for single stack DPDAs. We show in the full version of this paper the design of execution graphs for 2-Stack DPDAs and the matrix transition equations with respect to these graphs. The case of DPDAs with multiple stacks (more than two) follows readily.

6 Conclusion

In this paper, we present a (key-policy) attribute-based encryption scheme for deterministic multi-stack pushdown automata and, therefore, Turing machines, with polynomially bounded execution. Crucially, our scheme enjoys input-specific decryption time from standard cryptographic assumptions, in contrast to previous ABE schemes in which the decryption algorithms have always had worst-case runtime. We prove the security of our scheme based on the hardness of LWE problem in the selective security model.

An interesting open problem is to devise a technique whereby the benefits of input-specific complexity can be achieved for different classes of ABE, such as ABE for circuits. Another even more challenging problem is to design ABE schemes to handle a-priori unbounded automata or machines. The problem of handling non-deterministic Turing machines is also wide open.

Acknowledgement. The authors would like to thank Shweta Agrawal for insightful discussions, and the reviewers of ACNS 2016 for the helpful comments.

References

1. Agrawal, S., Boneh, D., Boyen, X.: Efficient lattice (H)IBE in the standard model. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 553–572. Springer, Heidelberg (2010)
2. Attrapadung, N.: Dual system encryption via doubly selective security: framework, fully secure functional encryption for regular languages, and more. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 557–577. Springer, Heidelberg (2014)
3. Boneh, D., Boyen, X.: Efficient selective identity-based encryption without random oracles. J. Cryptol. **24**(4), 659–693 (2011)
4. Boneh, D., Gentry, C., Gorbunov, S., Halevi, S., Nikolaenko, V., Segev, G., Vaikuntanathan, V., Vinayagamurthy, D.: Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 533–556. Springer, Heidelberg (2014)
5. Boyen, X.: Attribute-based functional encryption on lattices. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 122–142. Springer, Heidelberg (2013)
6. Boyen, X., Li, Q.: Attribute-based encryption for finite automata from LWE. In: Au, M.H., Miyaji, A. (eds.) ProvSec 2015. LNCS, vol. 9451, pp. 247–267. Springer, Heidelberg (2015)
7. Brakerski, Z., Langlois, A., Peikert, C., Regev, O., Stehlé, D.: Classical hardness of learning with errors. In: STOC 2013, pp. 575–584. ACM (2013)
8. Cash, D., Hofheinz, D., Kiltz, E., Peikert, C.: Bonsai trees, or how to delegate a lattice basis. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 523–552. Springer, Heidelberg (2010)
9. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: IJCAI 1991, vol. 1, pp. 331–337. Morgan Kaufmann Publishers Inc. (1991)

10. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: FOCS 2013, pp. 40–49. IEEE (2013)
11. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: STOC 2008, pp. 197–206. ACM (2008)
12. Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: How to run turing machines on encrypted data. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 536–553. Springer, Heidelberg (2013)
13. Gorbunov, S., Vaikuntanathan, V., Wee, H.: Attribute-based encryption for circuits. In: STOC 2013, pp. 545–554. ACM (2013)
14. Gorbunov, S., Vaikuntanathan, V., Wee, H.: Predicate encryption for circuits from LWE. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 503–523. Springer, Heidelberg (2015)
15. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: CCS 2006, pp. 89–98. ACM (2006)
16. Hohenberger, S., Waters, B.: Attribute-based encryption with fast decryption. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 162–179. Springer, Heidelberg (2013)
17. Katz, J., Sahai, A., Waters, B.: Predicate encryption supporting disjunctions, polynomial equations, and inner products. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 146–162. Springer, Heidelberg (2008)
18. Lewko, A., Waters, B.: Unbounded HIBE and attribute-based encryption. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 547–567. Springer, Heidelberg (2011)
19. Micciancio, D., Peikert, C.: Trapdoors for lattices: simpler, tighter, faster, smaller. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 700–718. Springer, Heidelberg (2012)
20. Okamoto, T., Takashima, K.: Adaptively attribute-hiding (hierarchical) inner product encryption. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 591–608. Springer, Heidelberg (2012)
21. Peikert, C.: Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In: STOC 2009, pp. 333–342. ACM (2009)
22. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: STOC 2005, pp. 84–93. ACM (2005)
23. Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 457–473. Springer, Heidelberg (2005)
24. Sipser, M.: Introduction to the Theory of Computation, vol. 2. Thomson Course Technology, Boston (2006)
25. Waters, B.: Ciphertext-policy attribute-based encryption: an expressive, efficient, and provably secure realization. In: Catalano, D., Fazio, N., Gennaro, R., Nicolosi, A. (eds.) PKC 2011. LNCS, vol. 6571, pp. 53–70. Springer, Heidelberg (2011)
26. Waters, B.: Functional encryption for regular languages. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 218–235. Springer, Heidelberg (2012)