The Power of Bamboo: On the Post-Compromise Security for Searchable Symmetric Encryption

Tianyang Chen[*,†,§], Peng Xu[✉,*,†,§], Stjepan Picek[††], Bo Luo[‖], Willy Susilo[**], Hai Jin[*,‡,§], Kaitai Liang[¶]
[*]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab
[†]Hubei Key Laboratory of Distributed System Security,
Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering
[‡]Cluster and Grid Computing Lab, School of Computer Science and Technology
[§]Huazhong University of Science and Technology, Wuhan, 430074, China
[††]Digital Security Group, Radboud University, Nijmegen, The Netherlands
[‖]Department of EECS and Institute of Information Sciences, The University of Kansas, Lawrence, KS, USA
[**]Institute of Cybersecurity and Cryptology, School of Computing and Information Technology,
University of Wollongong, Wollongong, NSW 2522, Australia
[¶]Faculty of Electrical Engineering, Mathematics and Computer Science,
Delft University of Technology, 2628 CD Delft, The Netherlands
{chentianyang, xupeng}@mail.hust.edu.cn, stjepan.picek@ru.nl, bluo@ku.edu,
wsusilo@uow.edu.au, hjin@hust.edu.cn, Kaitai.Liang@tudelft.nl

***Abstract*—Dynamic searchable symmetric encryption (DSSE) enables users to delegate the keyword search over dynamically updated encrypted databases to an honest-but-curious server without losing keyword privacy. This paper studies a new and practical security risk to DSSE, namely, *secret key compromise* (e.g., a user's secret key is leaked or stolen), which threatens all the security guarantees offered by existing DSSE schemes. To address this open problem, we introduce the notion of *searchable encryption with key-update* (SEKU) that provides users with the option of non-interactive key updates. We further define the notion of post-compromise secure with respect to leakage functions to study whether DSSE schemes can still provide data security after the client's secret key is compromised. We demonstrate that post-compromise security is achievable with a proposed protocol called "`Bamboo`". Interestingly, the leakage functions of `Bamboo` satisfy the requirements for both forward and backward security. We conduct a performance evaluation of `Bamboo` using a real-world dataset and compare its runtime efficiency with the existing forward-and-backward secure DSSE schemes. The result shows that `Bamboo` provides strong security with better or comparable performance.**

I. Introduction

A. Motivation

Dynamic searchable symmetric encryption (DSSE) [45], a type of structured encryption [20], [34], [43], is a cryptographic tool that enables a client to outsource its encrypted database (i.e., a collection of ciphertexts) to a remote server and further perform the dynamic data update and keyword search over the encrypted database, where the server is modeled as an honest-but-curious adversary. The client is allowed to maintain a secret key locally to generate secure keyword search queries and to issue data update queries to the server for adding/deleting ciphertexts to/from the encrypted database. DSSE provides secure queries over encrypted cloud-based databases and has been widely deployed in real-world applications, such as Lookout [53], bitglass [5], Cossack Labs' Acra [48], and MVISION Cloud [54].

Since the seminal work of DSSE by Kamara et al. [45], many research efforts have been devoted to constructing practical and efficient DSSE schemes. To provide high efficiency on data update and keyword search, DSSE discloses some information to the server, causing potential information leakage [24]. A well-studied solution to minimizing such information leakage is to use forward security [10], [63] to defend against attacks such as file-injection attack [72]. The idea behind forward security is to prevent data update queries from leaking the updated keywords. Bost et al. introduced another security notion, backward security [12], which limits the information leaked from the deleted ciphertexts during search queries. Since then, various DSSE schemes have been proposed to achieve both forward and backward security without loss of efficiency, e.g., [17], [18], [21], [25], [65], [66], [71], [73].

The security of all existing DSSE schemes relies on a "strong" assumption that the secret key of the client can always be protected and will not be compromised. Unfortunately, this assumption may not scale well in practice. For example, 23,000 secret keys of HTTPS certificates were compromised by network attackers [51], and Imperva inc. leaked customers' data after its cloud API key was stolen [57]. In the context of DSSE, if a client's secret key is exposed, the thief who obtains the key can easily break the encrypted database and observe

✉Peng Xu is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24725
www.ndss-symposium.org

data updates and search queries. In this paper, we pose the following question: *"Can a DSSE scheme still provide clients with data security while maintaining its performance efficiency if the client's secret key is compromised?"*

Due to the requirement of maintaining high performance, it may be infeasible to design a perfectly secure DSSE system that can protect a client's secret key at all times against various attacks. Hence we aim to increase the difficulty of attacking and limit the information leakage if the key is compromised. Our solution adds new features to DSSE that enable the clients to update the secret key based on their preferences. We would like to mention that this philosophy complies with the recommendations by the Data Security Standard of Payment Card Industry [22] and NIST Special Publication 800-57 [4].

B. Simple Solutions Do Not Work

To implement our design idea, we equip DSSE with a new and secure protocol (which we call KeyUpdate) to update the secret key of the encrypted database. A straightforward implementation of KeyUpdate is to mandate the client to download the entire encrypted database, decrypt the database, then re-encrypt it with a new secret key, and upload the re-encrypted database to the server. Clearly, this trivial solution will lead to huge bandwidth and client computation costs, which are linear to the size of the encrypted database. For a large-scale database, this solution is impractical. More importantly, it may fail to protect the security of ciphertexts generated in a "special time slot": the period between the key being compromised and the time when the new key is updated. Therefore, we require the proposed KeyUpdate protocol to deliver (1) efficiency: taking constant bandwidth and computational costs to delegate a secure KeyUpdate task to the server, and (2) security: ensuring the security of ciphertexts generated during the special time slot.

One potential approach is simply applying key-updatable tools to existing DSSE schemes for the KeyUpdate protocol with the desired efficiency and security. Unfortunately, this is very unlikely, particularly to the security requirement of guaranteeing the security of ciphertexts that are generated with the compromised keys. We take a close look at a pair of examples. One may adopt MITRA [18] to implement KeyUpdate by replacing the original PRF functions with those that enable key update, e.g., [8], [71]. However, if the secret key is compromised, the thief (with the key) can easily learn the content of a newly updated ciphertext by traversing all the possible keywords and counters. Similar vulnerability exists for ORION and HORUS [18]. One may also use *ciphertext-independent updatable encryption*, e.g., [13], [47], [50], to provide ORAM and OMAP [68] with key update function. However, this extension cannot guarantee the security of ciphertexts generated in the special time slot mentioned earlier because the compromised secret key remains valid to decrypt the updatable ORAM and OMAP until it is updated. A more detailed discussion can be found in Section VII-A.

C. Ideas Behind Bamboo

The proposed instance "Bamboo" implements both the traditional DSSE functions and an efficient KeyUpdate protocol that meets the two aforementioned requirements: efficiency and security. In this section, we explain how Bamboo protects the client's private data from being stolen while maintaining high search efficiency. The main ideas behind Bamboo are twofold: two-layer encryption and a hidden chain-like inter-ciphertext structure.

Bamboo uses a two-layer encryption mechanism to generate a ciphertext. The first layer (i.e., the inner layer) is used as a traditional encryption scheme, and the second one (i.e., the outer layer) is another encryption with the client's secret key designed for the key update. To generate a ciphertext, Bamboo first chooses a random number as the encryption key for the first layer to encrypt the original data. Then the encrypted data will be encrypted again in the second layer with the client's secret key.

The two-layer encryption mechanism guarantees that, even when the thief compromises the client's secret key and all the historical random numbers, the thief still cannot reveal any information from a newly generated ciphertext $\mathbf{C}$. Although the thief can decrypt the second layer encryption of $\mathbf{C}$ with the compromised client's secret key, s/he cannot decrypt the first layer in the absence of the random number that was used to generate $\mathbf{C}$. This feature enables Bamboo to maintain the security of the ciphertexts generated during the special time slot.

The second layer helps Bamboo to perform a non-interactive KeyUpdate protocol. The non-interactive feature reduces the overhead in executing the KeyUpdate protocol. The key-updatable feature guarantees that the thief cannot leverage a historically compromised secret key to decrypt the ciphertexts generated after the secret key is updated because the new secret key is unknown to the thief.

To gain efficient search performance, Bamboo employs a hidden chain-like inter-ciphertext structure to organize ciphertexts encrypted by the same keyword. Specifically, for any two successively generated ciphertexts encrypted by the same keyword, the latter encrypts an index and the random number used in the former one. Upon receiving a search query, the server can find a matching ciphertext and decrypt the index and the random number in the next ciphertext matching the same keyword. In the same way, the server can find all matching ciphertexts. Hence, Bamboo achieves sub-linear search efficiency as most practical DSSE schemes do.

D. Contributions

Bamboo is one instance of a new type of DSSE, *searchable encryption with key-update* (SEKU), that we introduced in this paper. SEKU captures all the functionalities provided by traditional DSSE and supports the non-trivial KeyUpdate protocol. We analyze the threat model, where the attacker could either be an honest-but-curious server or a malicious client who wants to steal the secret key of other clients and formalize post-compromise security via a common paradigm (i.e., leakage functions). The security guarantees that neither the honest-but-curious server nor the malicious client can learn any semantic information from data update and KeyUpdate queries. It subsumes forward security [10] and is compatible with backward security [12]. This means that a post-compromise secure SEKU instance can be both forward secure and backward secure.

TABLE I. COMPARISONS OF BAMBOO WITH RELATED FORWARD-AND-BACKWARD SECURE DSSE SCHEMES.

Scheme	KeyUpdate			Search		Data Update	Client
	Server Computation	Client Computation	Bandwidth	Computation	Bandwidth	Computation & Bandwidth	Storage
Fides [12]	$O(1)$	$O(N)$	$O(N)$	$O(a_w)$	$O(a_w)$	$O(1)$	$O(W\log F)$
Aura [65]	$O(1)$	$O(N)$	$O(N)$	$O(n_w)$	$O(n_w)$	$O(1)$	$O(Wd_{\max})$
$\mathtt{SD}_a$ [25]	$O(1)$	$O(N)$	$O(N)$	$O(a_w+\log N)$	$O(a_w+\log N)$	$O(\log N)$	$O(1)$
$\mathtt{SD}_d$ [25]	$O(1)$	$O(N)$	$O(N)$	$O(a_w+\log N)$	$O(a_w+\log N)$	$O(\log^3 N)$	$O(1)$
MITRA [18]	$O(N)$	$O(1)$	$O(1)$	$O(a_w)$	$O(a_w)$	$O(1)$	$O(W\log F)$
Bamboo (Ours)	$O(N)$	$O(1)$	$O(1)$	$O(a_w)$	$O(a_{\max})$	$O(1)$	$O(W\log F)$

The KeyUpdate costs of Fides, Aura, $\mathtt{SD}_a$, and $\mathtt{SD}_d$ are counted from the trivially implemented KeyUpdate, and the KeyUpdate of MITRA is implemented by replacing the original PRF with the key-updatable PRF [71]. N denotes the total number of keyword-and-file-identifier pairs, W denotes the number of distinct keywords, F denotes the number of files, and $d_{\max}$ denotes the allowed maximum number of deletion queries. For keyword w, $a_{\max}$ is the padding constant used for hiding the real search result size, a_w is the total number of data update queries the client has issued, and n_w is the number of files containing w. All the listed schemes achieve $O(N)$ storage complexity on the server side.

We use the first SEKU scheme Bamboo to illustrate the above ideas and prove that Bamboo is post-compromise secure. The leakage functions of Bamboo satisfy the backward security requirements. Bamboo achieves an excellent balance between security and performance compared to the existing DSSE schemes that offer both forward security and the same level of backward security, as shown in Table I. In terms of KeyUpdate complexity, only Bamboo and MITRA can delegate key update operations to the server and save a considerable amount of client computation and bandwidth costs compared to others, like Fides, Aura, $\mathtt{SD}_d$, and $\mathtt{SD}_a$. For the search operation, Bamboo achieves the same level of computation as Fides and MITRA and outperforms $\mathtt{SD}_a$ and $\mathtt{SD}_d$. The bandwidth cost of Bamboo in searching a keyword is dominated by a maximum padding value $a_{\max}$ (defined by the client in practice). $a_{\max}$ helps Bamboo to achieve post-compromise security and we will elaborate in Section V-E how to reduce the search bandwidth by adaptively adjusting the padding value. Bamboo provides the same data update complexity as Fides, Aura, and MITRA, and better performance than $\mathtt{SD}_a$ and $\mathtt{SD}_d$. Finally, Bamboo has the same storage complexity as the other schemes on the server side; the client storage cost is on par with Fides and MITRA and less than that in Aura, but a little higher than $\mathtt{SD}_a$ and $\mathtt{SD}_d$. The experimental results show that the storage costs of Bamboo on both the server and client sides are practical (see Section VI-E).

We conduct a comprehensive evaluation of the performance of Bamboo using a real-world dataset (which is extracted from Wikipedia) and by comparing with Fides, Aura, and MITRA. The results show that Bamboo outperforms the state-of-the-art schemes. For example, when issuing a data update query, Bamboo saves 97.22% and 68.33% on client time costs compared to Aura and Fides, respectively. When running KeyUpdate with 300 milliseconds network delay, Bamboo is 3.66 times and 2.57 times faster than Fides and Aura, respectively; and the client time cost is much smaller. In terms of search performance, Bamboo outperforms Fides and significantly reduces client time costs as compared to both Fides and MITRA.

II. THREAT MODEL OF SEKU

SEKU should protect a client's encrypted database against two types of probabilistic polynomial time (PPT) adversaries. One is widely recognized in the setting of DSSE, namely the honest-but-curious server, and the other, which is new to DSSE, is called the thief who steals the secret key.

Server. Much like the traditional DSSE server, a SEKU server can store the encrypted database for a SEKU client, add/delete ciphertexts to/from the database, and further perform a secure keyword search. Beyond that, SEKU enables the server to execute the key update, which is delegated by the client so that the secret key of the encrypted database can be updated to a new one. The SEKU server here is honest while executing all predefined operations; meanwhile, it can observe the information leakage from those operations.

Thief. We allow this party to compromise the client's secret key, eavesdrop on the communication between the client and the server, and obtain a copy of the encrypted database. We do not assume that this party takes over the client or issues queries to the server with the compromised key. We notice that some DSSE schemes may need the client to store extra secret information about the encrypted database (i.e., the private state) along with the secret key. We here assume the secret information will also be leaked if the secret key is compromised.

SEKU provides two types of threat models based on whether there is collusion between the server and the thief (i.e., sharing their information). The essential difference between these types is the adversarial ability of the server. Specifically, collusion enables the server to learn the client's secret key so that it is no longer an honest-but-curious server but a fully malicious adversary. This malicious server can reveal all the contents of the encrypted database with the key. In this case, the security goal and the corresponding secure construction roadmaps will completely differ from the honest-but-curious case.

Type 1: The Moderate Threat Model. The server and the thief are not allowed to collude. The thief can be given the client's secret key and observe public parameters, data update queries, and the encrypted database. While the client issues keyword search or KeyUpdate queries to the server, the thief can observe the communications between the client and the server. Figure 1 outlines what the server and the thief can observe in this model.

Type 2: The Stronger Threat Model. Collusion between the two parties is allowed such that they can share exactly the same views as shown in Figure 1. This makes the server so powerful that the following *technical impossibilities* will incur.

• *Non-interactive KeyUpdate design.* In Type 2 threat model, the server is regarded as malicious and can use the compromised secret key of the client to break all the ciphertexts which were generated before the key compromise. In this

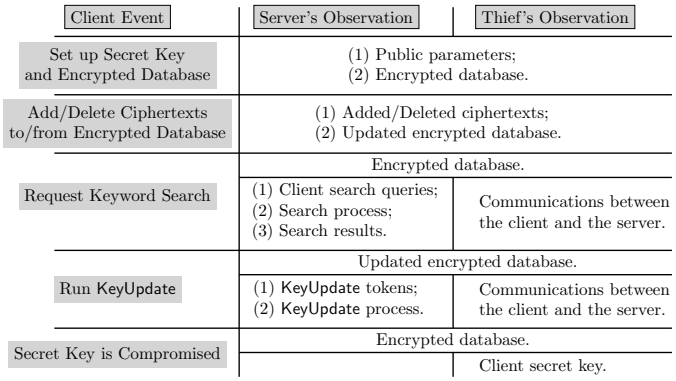


Fig. 1. The Observations on Client Queries and Key Compromise in the Moderate Threat Model. Note that in the stronger threat model, the two parties will share their observations.

case, a secure KeyUpdate must hide the relationships between the pre-key-updated and post-key-updated ciphertexts from the server to avoid it leverages the knowledge of the pre-key-updated ciphertexts to infer information from the post-key-updated ones. In other words, the KeyUpdate must update the encryption key of the encrypted database obliviously. It is nearly impossible to design such a non-interactive KeyUpdate. We notice that indistinguishability obfuscation ($i\mathcal{O}$) [33], [40] could be a potential solution to offload the oblivious KeyUpdate to the server. However, it is still unknown how to construct a practical $i\mathcal{O}$ scheme.

• *Interactive KeyUpdate design.* The simple solution described in Section I-B may achieve the oblivious KeyUpdate, but clearly, it is interactive and expensive. Even if there may exist a practical and efficient interactive key update approach, there are still further concerns and impossibilities for the design. Due to page limit, we leave the details in Section VII-B.

Because of these impossibilities, this work focuses on Type 1 model. We will define the security notions as the indistinguishability of one real and one simulated SEKU in Section III, and formulate the leakage functions of post-compromise security in Section IV.

III. SEKU and Its Security Definitions

A. Notations

Let $\mathcal{A}_{\text{Srv}}$ and $\mathcal{A}_{\text{Thf}}$ denote the server and the thief, respectively. Let $\lambda \in \mathbb{N}$ denote the security parameter. We use $e \xleftarrow{\$} \mathcal{X}$ to denote uniformly sampling an element e from a distribution or set $\mathcal{X}$. For a set $\mathcal{X}$, $|\mathcal{X}|$ is the total number of elements in $\mathcal{X}$. $\{0,1\}^n$ $(n \in \mathbb{N})$ denotes the set of all $n-$bit strings. $s_1||s_2$ represents the concatenation of two strings s_1 and s_2. Let $\mathcal{W} = \{0,1\}^k$ $(k \in \mathbb{N})$ be the keyword space and $\mathcal{ID} = \{0,1\}^{\lambda-1} \setminus \{0^{\lambda-1}\}$ be the file identifier space, where we assume that the string with $(\lambda-1)$-bit zeros, $0^{\lambda-1}$ will never be used as a valid file identifier. We use the term *entry* to denote the tuple $(op,(w,id))$ of an addition or deletion operation $op \in \{add, del\}$, and a pair of a keyword $w \in \mathcal{W}$ and a file-identifier $id \in \mathcal{ID}$.

B. SEKU Syntax

Definition 1 (SEKU)**.** *A SEKU scheme Σ is composed of four protocols Setup, DataUpdate, Search, and KeyUpdate defined as:*

- *Setup*(λ) *takes as input the security parameter λ, generates the secret key K_Σ and private state $\mathbf{State}$ for the client, and initializes the encrypted database $\mathbf{EDB}$ for the server.*
- *DataUpdate*$(K_\Sigma, \mathbf{State}, op, (w,id); \mathbf{EDB})$ *takes as input the secret key K_Σ and private state $\mathbf{State}$ to encrypt the entry $(op,(w,id))$ from the client, and finally stores the generated ciphertext into the encrypted database $\mathbf{EDB}$.*
- *Search*$(K_\Sigma, \mathbf{State}, w; \mathbf{EDB})$ *takes as input the secret key K_Σ, private state $\mathbf{State}$, and a keyword w from the client, and securely delegates the search query of w over the encrypted database $\mathbf{EDB}$ to the server. Finally, this protocol outputs the search results.*
- *KeyUpdate*$(K_\Sigma, \mathbf{State}; \mathbf{EDB})$ *takes as input the secret key K_Σ and private state $\mathbf{State}$ from the client, and delegates the key update query to the server. During this protocol, the client updates the secret key K_Σ to a new secret key K'_Σ, and the server executes the key update over $\mathbf{EDB}$ to update the encryption key K_Σ of all the stored ciphertexts to K'_Σ.*

Correctness. We say a SEKU scheme is correct if for any $\lambda \in \mathbb{N}$ and $(K_\Sigma, \mathbf{State}; \mathbf{EDB}) \leftarrow \mathsf{Setup}(\lambda)$, for any $poly(\lambda)$ executions of $\mathsf{DataUpdate}(K_\Sigma, \mathbf{State}, op, (w,id); \mathbf{EDB})$, $\mathsf{Search}(K_\Sigma, \mathbf{State}, w; \mathbf{EDB})$, and $\mathsf{KeyUpdate}(K_\Sigma, \mathbf{State}; \mathbf{EDB})$, protocol $\mathsf{Search}(K_\Sigma, \mathbf{State}, w; \mathbf{EDB})$ always returns the set of file identifiers paired with the specific keyword w that have been inserted into $\mathbf{EDB}$ by executing $\mathsf{DataUpdate}(K_\Sigma, \mathbf{State}, op = add, (w,id); \mathbf{EDB})$ and not yet deleted by executing $\mathsf{DataUpdate}(K_\Sigma, \mathbf{State}, op = del, (w,id); \mathbf{EDB})$.

Based on Type 1 model, we define the SEKU security against the server and the thief as the indistinguishability of a real and a simulated game. In the real game, the adversary (namely, the server or the thief) runs a real SEKU scheme with adaptively selected inputs, while in the simulated game, the adversary plays with a simulator that simulates a SEKU scheme with a set of leakage functions as inputs. The leakage functions define what information the adversary can infer from observing a real SEKU scheme. If the real game is indistinguishable from the simulated game in the view of the adversary, we say that the leakage of the SEKU scheme to the adversary is strictly bounded by the leakage functions. The following subsections apply the above ideas to formally define the adaptive security of SEKU against the server $\mathcal{A}_{\text{Srv}}$ and the thief $\mathcal{A}_{\text{Thf}}$, respectively.

C. Adaptive Security against $\mathcal{A}_{Srv}$

We require that there exists a simulator $\mathcal{S}$ to simulate an ideal game. In the ideal game, server $\mathcal{A}_{\text{Srv}}$ can adaptively issue queries of Setup, DataUpdate, Search, and KeyUpdate, and simulator $\mathcal{S}$ forges the corresponding responses with leakage functions $\mathcal{L}^{Stp}_{\text{Srv}}, \mathcal{L}^{DaUpdt}_{\text{Srv}}, \mathcal{L}^{Srch}_{\text{Srv}}$, and $\mathcal{L}^{KeyUpdt}_{\text{Srv}}$. We say that

SEKU is adaptively secure against $\mathcal{A}_{\text{Srv}}$ if the ideal game is indistinguishable from a real game in the view of $\mathcal{A}_{\text{Srv}}$. Hence, we have the following formal definition.

Definition 2 (Adaptive Security Against $\mathcal{A}_{\text{Srv}}$)**.** *Given leakage functions $\mathcal{L}_{Srv} = (\mathcal{L}_{Srv}^{Stp}, \mathcal{L}_{Srv}^{DaUpdt}, \mathcal{L}_{Srv}^{Srch}, \mathcal{L}_{Srv}^{KeyUpdt})$, a SEKU scheme Σ is said to be $\mathcal{L}_{Srv}$-adaptively secure if for any sufficiently large security parameter $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}_{Srv}$, there exists an efficient simulator $\mathcal{S} = (\mathcal{S}.\mathsf{Setup}, \mathcal{S}.\mathsf{DataUpdate}, \mathcal{S}.\mathsf{Search}, \mathcal{S}.\mathsf{KeyUpdate})$, such that the probability $|Pr[Real_{\mathcal{A}_{Srv}}^{\Sigma}(\lambda) = 1] - Pr[Ideal_{\mathcal{A}_{Srv},\mathcal{S},\mathcal{L}_{Srv}}^{\Sigma}(\lambda) = 1]|$ is negligible in λ, where games $Real_{\mathcal{A}_{Srv}}^{\Sigma}(\lambda)$ and $Ideal_{\mathcal{A}_{Srv},\mathcal{S},\mathcal{L}_{Srv}}^{\Sigma}(\lambda)$ are defined as:*

- $Real_{\mathcal{A}_{Srv}}^{\Sigma}(\lambda)$*: This real game implements all real SEKU protocols. After initializing Σ by running protocol* Setup*, $\mathcal{A}_{Srv}$ adaptively issues* DataUpdate*,* Search*, and* KeyUpdate *queries, and observes the real transcripts of those queries. In the end, $\mathcal{A}_{Srv}$ outputs one bit.*
- $Ideal_{\mathcal{A}_{Srv},\mathcal{S},\mathcal{L}_{Srv}}^{\Sigma}(\lambda)$*: In this game, $\mathcal{A}_{Srv}$ interacts with simulator $\mathcal{S}$ and issues the same queries as in the real game. Simulator $\mathcal{S}$ takes the leakage functions $\mathcal{L}_{Srv} = (\mathcal{L}_{Srv}^{Stp}, \mathcal{L}_{Srv}^{DaUpdt}, \mathcal{L}_{Srv}^{Srch}, \mathcal{L}_{Srv}^{KeyUpdt})$ as inputs and respectively simulates the corresponding transcripts of SEKU protocols* Setup*,* DataUpdate*,* Search*, and* KeyUpdate *for $\mathcal{A}_{Srv}$ by running $\mathcal{S}$.*Setup*, $\mathcal{S}$.*DataUpdate*, $\mathcal{S}$.*Search*, and $\mathcal{S}$.*KeyUpdate*. In the end, $\mathcal{A}_{Srv}$ outputs one bit.*

D. Adaptive Security against $\mathcal{A}_{Thf}$

This security is also defined as the indistinguishability between a real game and an ideal game. Specifically, we require there exists a simulator $\mathcal{S}'$ to simulate the ideal game. In the simulation process, $\mathcal{S}'$ simulates protocols Setup, DataUpdate, Search, and KeyUpdate with leakage functions $\mathcal{L}_{\text{Thf}}^{Stp}$, $\mathcal{L}_{\text{Thf}}^{DaUpdt}$, $\mathcal{L}_{\text{Thf}}^{Srch}$, and $\mathcal{L}_{\text{Thf}}^{KeyUpdt}$, respectively. Besides, $\mathcal{S}'$ forges the key compromise event with leakage function $\mathcal{L}_{\text{Thf}}^{KeyLeak}$, where $\mathcal{L}_{\text{Thf}}^{KeyLeak}$ is from a real key-compromise event. We say that SEKU is adaptively secure against $\mathcal{A}_{\text{Thf}}$ if the ideal game is indistinguishable from a real game in the view of $\mathcal{A}_{\text{Thf}}$. Formally, we have:

Definition 3 (Adaptive Security Against $\mathcal{A}_{\text{Thf}}$)**.** *Given leakage functions $\mathcal{L}_{Thf} = (\mathcal{L}_{Thf}^{Stp}, \mathcal{L}_{Thf}^{DaUpdt}, \mathcal{L}_{Thf}^{Srch}, \mathcal{L}_{Thf}^{KeyUpdt}, \mathcal{L}_{Thf}^{KeyLeak})$, a SEKU scheme Σ is $\mathcal{L}_{Thf}$-adaptively secure if for any sufficiently large security parameter $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}_{Thf}$, there exists an efficient simulator $\mathcal{S}' = (\mathcal{S}'.\mathsf{Setup}, \mathcal{S}'.\mathsf{DataUpdate}, \mathcal{S}'.\mathsf{Search}, \mathcal{S}'.\mathsf{KeyUpdate}, \mathcal{S}'.\mathsf{KeyLeak})$, such that the probability $|Pr[Real_{\mathcal{A}_{Thf}}^{\Sigma}(\lambda) = 1] - Pr[Ideal_{\mathcal{A}_{Thf},\mathcal{S}',\mathcal{L}_{Thf}}^{\Sigma}(\lambda) = 1]|$ is negligible in λ, where games $Real_{\mathcal{A}_{Thf}}^{\Sigma}(\lambda)$ and $Ideal_{\mathcal{A}_{Thf},\mathcal{S}',\mathcal{L}_{Thf}}^{\Sigma}(\lambda)$ are defined as:*

- $Real_{\mathcal{A}_{Thf}}^{\Sigma}(\lambda)$*: This real game exactly implements all SEKU protocols. The thief $\mathcal{A}_{Thf}$ adaptively issues* DataUpdate*,* Search*, and* KeyUpdate *queries and observes the real transcripts generated by those queries. In addition, $\mathcal{A}_{Thf}$ can adaptively compromise the secret key and private state multiple times. In the end, it outputs one bit.*
- $Ideal_{\mathcal{A}_{Thf},\mathcal{S}',\mathcal{L}_{Thf}}^{\Sigma}(\lambda)$*: In this game, the thief $\mathcal{A}_{Thf}$ issues the same queries as in the real game. $\mathcal{S}'$ forges the corresponding transcripts for $\mathcal{A}_{Thf}$ by running $\mathcal{S}'$.*Setup*, $\mathcal{S}'$.*DataUpdate*, $\mathcal{S}'$.*Search*, and $\mathcal{S}'$.*KeyUpdate *with leakage functions $\mathcal{L}_{Thf}^{Stp}$, $\mathcal{L}_{Thf}^{DaUpdt}$, $\mathcal{L}_{Thf}^{Srch}$, and $\mathcal{L}_{Thf}^{KeyUpdt}$, respectively. When a real key-compromise event happens, $\mathcal{S}'$ runs $\mathcal{S}'$.*KeyLeak *with leakage function $\mathcal{L}_{Thf}^{KeyLeak}$ to simulate the key-compromise event. In the end, $\mathcal{A}_{Thf}$ outputs one bit.*

IV. Post-Compromise Security of SEKU

A. Overview

As explained in Section II, this work focuses on Type 1 security model where the server and the thief do not collude. In this section, we will define the *post-compromise security* of SEKU against Type 1 model by specifying the information leakage allowed to the thief $\mathcal{A}_{\text{Thf}}$ and the server $\mathcal{A}_{\text{Srv}}$. To resist the thief $\mathcal{A}_{\text{Thf}}$, the security must guarantee the privacy of both the ciphertexts and search queries that are generated with non-compromised secret keys, even if any of the historical and future secret keys are compromised. To achieve this goal, we disallow $\mathcal{A}_{\text{Thf}}$ to learn anything from protocol KeyUpdate. In addition, we also expect that the post-compromise security maintains the confidentiality of the newly generated ciphertexts against $\mathcal{A}_{\text{Thf}}$, even if those ciphertexts are generated with a compromised key.

When executing protocol KeyUpdate, the server $\mathcal{A}_{\text{Srv}}$ is delegated to update the secret key of ciphertexts to a new one. It is necessary to guarantee that protocol KeyUpdate does not leak any privacy to $\mathcal{A}_{\text{Srv}}$. In other words, the post-compromise security requires that KeyUpdate leaks nothing to $\mathcal{A}_{\text{Srv}}$, except that $\mathcal{A}_{\text{Srv}}$ can know if a ciphertext is key-updated from an old (existing) ciphertext.

In summary, our security defines the following basic limitations on information leakage:

- **Limit 1**: A compromised key is allowed only to break the ciphertexts that were generated since the last execution of protocol KeyUpdate.
- **Limit 2**: Both a KeyUpdate query and its resulting encrypted database leak nothing to the thief $\mathcal{A}_{\text{Thf}}$.
- **Limit 3**: The thief $\mathcal{A}_{\text{Thf}}$ cannot distinguish any two Search queries.
- **Limit 4**: A DataUpdate query leaks nothing to the thief $\mathcal{A}_{\text{Thf}}$. Note this implies that a DataUpdate query also leaks nothing to the server $\mathcal{A}_{\text{Srv}}$.
- **Limit 5**: Protocol KeyUpdate leaks (at most) if a ciphertext is key-updated from an old (existing) ciphertext to the server $\mathcal{A}_{\text{Srv}}$.

We note that the security does not restrict the information leakage of Search queries to $\mathcal{A}_{\text{Srv}}$. One may design a SEKU instance with the expected information leakage of a Search query to satisfy the security requirements in practice.

B. Definition of Timestamp

In SEKU, a timestamp denotes when a query is issued. Let Q^{DU}, Q^{Srch}, and Q^{KU} denote three client query lists: (1) Q^{DU} contains all DataUpdate queries in the form of $(u, op, (w, id))$, (2) Q^{Srch} records all Search queries in the form of (u, w), and (3) Q^{KU} stores all KeyUpdate timestamps in the form of u, where $u \in \mathbb{N}$, $op \in \{add, del\}$, $w \in \mathcal{W}$, and $id \in \mathcal{ID}$. We define U_{now} as the timestamp of a current query.

Since all timestamps are unique, we can use a timestamp to identify a ciphertext. For example, given the ciphertext $\mathbf{C}$ generated by a DataUpdate query $(u, op, (w, id)) \in Q^{\text{DU}}$, we can use the timestamp u to identify the ciphertext $\mathbf{C}$. Given a KeyUpdate query issued at timestamp u', suppose this query updates n existing ciphertexts $(\mathbf{C}_1, ..., \mathbf{C}_n)$ and then inserts the resulted ciphertexts $(\mathbf{C}'_1, ..., \mathbf{C}'_n)$ to $\mathbf{EDB}$, we associate ciphertext $\mathbf{C}'_i$ $(i \in [1, n])$ with timestamp $u' + i - 1$, and the subsequent queries will start at timestamp $u' + n$.

C. The Formal Definition

We first define several basic leakage functions and then use them to define post-compromise security.

Basic Leakage Functions. Let $u = \mathsf{Time}(\mathbf{C})$ denote retrieving the corresponding timestamp u of a given ciphertext $\mathbf{C}$. We use $\mathsf{CTRelation}(u^{\text{KU}})$ to denote the timestamp relationships between the pre-key-updated ciphertexts and the post-key-updated ciphertexts after completing a KeyUpdate query at timestamp $u^{\text{KU}} \in Q^{\text{KU}}$. Formally, we have

$$\mathsf{CTRelation}(u^{\text{KU}}) = \{(u, u') \mid \exists \text{ ciphertexts } \mathbf{C} \text{ and } \mathbf{C}',\ u = \mathsf{Time}(\mathbf{C}) \text{ and } u' = \mathsf{Time}(\mathbf{C}') \text{ s.t. } \mathbf{C} \text{ is updated from } \mathbf{C}' \text{ by executing KeyUpdate at timestamp } u^{\text{KU}}\}.$$

Let $\mathsf{KUHist}(u)$ record each KeyUpdate timestamp u^{KU} no more than u and the corresponding $\mathsf{CTRelation}(u^{\text{KU}})$, namely

$$\mathsf{KUHist}(u) = \{(u^{\text{KU}}, \mathsf{CTRelation}(u^{\text{KU}})) \mid u^{\text{KU}} \in Q^{\text{KU}} \text{ and } u^{\text{KU}} \le u\}.$$

Let $\mathsf{CUHist}(u)$ be the original data of all the ciphertexts generated by executing the KeyUpdate query at the maximum timestamp u^{KU} satisfying $u^{\text{KU}} \le u$. If no such u^{KU}, $\mathsf{CUHist}(u) = \emptyset$. Formally, we have

$$\mathsf{CUHist}(u) = \{(u', op, (w, id)) \mid \exists \text{ ciphertext } \mathbf{C},\ u' = \mathsf{Time}(\mathbf{C}) \text{ and } (op, (w, id)) \text{ is the content of } \mathbf{C} \text{ s.t. } \mathbf{C} \text{ is generated by executing KeyUpdate at timestamp } u^{\text{KU}} \text{ and } u^{\text{KU}} \text{ satisfying } u^{\text{KU}} \le u \text{ is the maximum one in } Q^{\text{KU}}\}.$$

Let $\mathsf{DUHist}(u)$ denote all the DataUpdate queries issued by the client since the KeyUpdate query of the maximum timestamp u^{KU}, where u^{KU} satisfies $u^{\text{KU}} \le u$. If no such u^{KU}, we define $u^{\text{KU}} = 0$. Formally, We have

$$\mathsf{DUHist}(u) = \{(u', op, (w, id)) \mid (u', op, (w, id)) \in Q^{\text{DU}} \text{ s.t. } u' > u^{\text{KU}} \text{ where } u^{\text{KU}} \text{satisfying } u^{\text{KU}} \le u \text{ is the maximum one in } Q^{\text{KU}} \cup \{0\}\}.$$

Next, we define the leakage functions of post-compromise security according to the limitations given in Section IV-A.

Leakage Functions of Key-Compromise. According to **Limit 1**, when the secret key is compromised at timestamp u, $\mathcal{A}_{\text{Thf}}$ is only allowed to learn the content of encrypted database $\mathbf{EDB}$ and client queries since the last execution of protocol KeyUpdate. The leakage function $\mathcal{L}_{\text{Thf}}^{KeyLeak}$ is defined as

$$\mathcal{L}_{\text{Thf}}^{KeyLeak} = \mathcal{L}'_{\text{Thf}}(\mathsf{CUHist}(U_{\text{now}}), \mathsf{DUHist}(U_{\text{now}}))$$

where $\mathcal{L}'_{\text{Thf}}$ is a stateless function.

Leakage Functions of KeyUpdate. According to **Limit 2** and **Limit 5**, during the KeyUpdate process, the thief cannot obtain any information, and the server can only learn the timestamp relationships between the pre-key-updated ciphertexts and the post-key-updated ciphertexts. We have

$$\mathcal{L}_{\text{Thf}}^{KeyUpdt} = \text{NULL} \text{ and } \mathcal{L}_{\text{Srv}}^{KeyUpdt} = \mathcal{L}'_{\text{Srv}}(\mathsf{KUHist}(U_{\text{now}}))$$

where $\mathcal{L}'_{\text{Srv}}$ is a stateless function.

Leakage Functions of Search. According to **Limit 3**, for any two keywords w_1 and w_2, $\mathcal{L}_{\text{Thf}}^{Srch}(w_1)$ should be indistinguishable from $\mathcal{L}_{\text{Thf}}^{Srch}(w_2)$. Namely, the thief cannot obtain any information from the Search process, even if it has compromised the secret key. Thus, the leakage function $\mathcal{L}_{\text{Thf}}^{Srch}$ is defined as

$$\mathcal{L}_{\text{Thf}}^{Srch}(w) = \text{NULL}.$$

Leakage functions of DataUpdate. According to **Limit 4**, a newly issued DataUpdate query should leak nothing to both the thief and the server. We define $\mathcal{L}_{\text{Thf}}^{DaUpdt}$ and $\mathcal{L}_{\text{Srv}}^{DaUpdt}$ as

$$\mathcal{L}_{\text{Thf}}^{DaUpdt}(op, (w, id)) = \text{NULL},$$
$$\mathcal{L}_{\text{Srv}}^{DaUpdt}(op, (w, id)) = \text{NULL}.$$

Finally, according to the adaptive security against the thief and the server defined in Section III and the above-defined leakage functions, we formalize the post-compromise security as below.

Definition 4 (Post-Compromise Security)**.** *A SEKU scheme is post-compromise secure* iff *it is $\mathcal{L}_{Srv}$-adaptively secure and $\mathcal{L}_{Thf}$-adaptively secure with the following restrictions on the leakage functions $\mathcal{L}_{Srv}$ and $\mathcal{L}_{Thf}$ simultaneously:*

1) *For $\mathcal{L}_{Srv} = (\mathcal{L}_{Srv}^{Stp}, \mathcal{L}_{Srv}^{DaUpdt}, \mathcal{L}_{Srv}^{Srch}, \mathcal{L}_{Srv}^{KeyUpdt})$, leakage functions $\mathcal{L}_{Srv}^{DaUpdt}$ and $\mathcal{L}_{Srv}^{KeyUpdt}$ can be written as:*

$$\mathcal{L}_{Srv}^{DaUpdt}(op, (w, id)) = NULL,$$
$$\mathcal{L}_{Srv}^{KeyUpdt} = \mathcal{L}'_{Srv}(\mathsf{KUHist}(U_{now})),$$

where $\mathcal{L}'_{Srv}$ is a stateless function.

2) *For $\mathcal{L}_{Thf} = (\mathcal{L}_{Thf}^{Stp}, \mathcal{L}_{Thf}^{DaUpdt}, \mathcal{L}_{Thf}^{Srch}, \mathcal{L}_{Thf}^{KeyUpdt}, \mathcal{L}_{Thf}^{KeyLeak})$, leakage functions $\mathcal{L}_{Thf}^{DaUpdt}$, $\mathcal{L}_{Thf}^{Srch}$, $\mathcal{L}_{Thf}^{KeyUpdt}$, and $\mathcal{L}_{Thf}^{KeyLeak}$ can be written as:*

$$\mathcal{L}_{Thf}^{DaUpdt}(op, (w, id)) = NULL,$$
$$\mathcal{L}_{Thf}^{Srch}(w) = NULL,\ \mathcal{L}_{Thf}^{KeyUpdt} = NULL,$$
$$\mathcal{L}_{Thf}^{KeyLeak} = \mathcal{L}'_{Thf}(\mathsf{CUHist}(U_{now}), \mathsf{DUHist}(U_{now})),$$

where $\mathcal{L}'_{Thf}$ is a stateless function.

Note that the post-compromise security does not put any restriction on leakage functions $\mathcal{L}_{Srv}^{Stp}$, $\mathcal{L}_{Srv}^{Srch}$, and $\mathcal{L}_{Thf}^{Stp}$.

The definition of leakage function $\mathcal{L}_{\text{Srv}}^{DaUpdt}$ in the post-compromise security also satisfies the notion of forward security in the context of DSSE [10], [63]. Specifically, the forward security requires that *a DataUpdate query does not leak any information about the updated keywords*. The post-compromise security defines the leakage functions of DataUpdate for both $\mathcal{A}_{\text{Srv}}$ and $\mathcal{A}_{\text{Thf}}$ are NULL. Such a definition clearly satisfies the forward security. Hence, we state that post-compromise security subsumes forward security. It is worth mentioning that the post-compromise security does not explicitly limit the leakage function $\mathcal{L}_{\text{Srv}}^{Srch}$. One may design a SEKU instance with different $\mathcal{L}_{\text{Srv}}^{Srch}$ to achieve various strengths of security against the server, e.g., backward security. Thus, post-compromise security is compatible with backward security.

V. Bamboo: A SEKU Instance

This section presents Bamboo, a post-compromise-secure SEKU instance that achieves constant DataUpdate complexity, sub-linear Search overhead, and non-interactive KeyUpdate. Bamboo leverages the two-layer encryption and inter-ciphertext chain-like structure techniques to generate its ciphertexts.

A. Building Blocks

The construction of Bamboo relies on an invertible mapping function and the Diffie-Hellman key exchange protocol. This part briefly introduces these building blocks.

Invertible Mapping Function. The invertible mapping function $\pi : \{0,1\}^n \to \mathbb{G}$ probabilistically maps an $n-$bit string to an element of a multiplicative cyclic group $\mathbb{G}$ of prime order q, and π^{-1} denotes the deterministic inverse of π. We use $(n, \mathbb{G}, q, \pi, \pi^{-1}) \leftarrow \mathbf{PGen}(\lambda, n)$ to denote an efficient probabilistic algorithm that takes as input a security parameter λ and bit length n of strings, and initializes the invertible mapping function π and its inverse π^{-1}. We assume that the DDH assumption [7] (defined in Appendix A) holds in the group $\mathbb{G}$. Boyd et al. [13] explained how to implement such probabilistic π by embedding a bit string into the X-coordinate of an elliptic curve point.

Diffie-Hellman Key Exchange Protocol [28]. This protocol enables the client and the server to generate a shared key via an insecure communication channel without prior shared secrets. We briefly introduce the elliptic curve variant of the Diffie-Hellman key exchange protocol. The client and the server first initialize an elliptic curve group $\mathbb{G}'$ whose order is prime q'. Let g' be a generator of $\mathbb{G}'$. Then, the client samples a secret random number $a \in \mathbb{Z}_{q'}^*$ and the server chooses a secret random number $b \in \mathbb{Z}_{q'}^*$. Next, the client and the server compute and exchange g'^a and g'^b, respectively. Finally, they compute the shared key $(g'^b)^a = (g'^a)^b$. After that, they can use the shared key to establish a secure channel. In the construction of Bamboo, when running the key change protocol, we assume the secret random numbers and the shared key are ephemeral. Namely, they will be permanently deleted after the secure channel is closed and will never be leaked out the memories of the client and the server.

Algorithm 1 Protocols Setup and DataUpdate of Bamboo.

Setup$(\lambda, a_{\max})$

1: Initialize an invertible mapping function and its inverse $(\lambda, \mathbb{G}, q, \pi, \pi^{-1}) \leftarrow \mathbf{PGen}(\lambda, \lambda)$
2: Initialize three cryptographic hash functions $\mathbf{H}_1 : \{0,1\}^* \to \mathbb{G}$, $\mathbf{H}_2 : \{0,1\}^* \to \mathbb{G}$, and $\mathbf{G} : \{0,1\}^* \to \mathbb{G}$
3: Initialize the Diffie-Hellman key exchange protocol parameters including the elliptic curve group $\mathbb{G}'$ of prime order q' and a generator $g' \in \mathbb{G}'$
4: Initialize two empty maps $\mathbf{State} \leftarrow \emptyset$ and $\mathbf{EDB} \leftarrow \emptyset$
5: Initialize the secret key $K_\Sigma = (K_1, K_2) \xleftarrow{\$} \mathbb{Z}_q^* \times \mathbb{Z}_q^*$
6: Send the encrypted database $\mathbf{EDB}$ to the server

DataUpdate$(K_\Sigma, \mathbf{State}, op, (w, id); \mathbf{EDB})$

Client:

1: Retrieve record (tk_w, cnt_w) from $\mathbf{State}[w]$
2: **if** $(tk_w, cnt_w) = (\text{NULL}, \text{NULL})$ **then**
3: Randomly draw $tk_w \leftarrow \{0,1\}^\lambda$ and initialize $cnt_w \leftarrow 0$
4: **end if**
5: Accumulate $cnt_w \leftarrow cnt_w + 1$
6: Randomly sample $tk'_w \leftarrow \{0,1\}^\lambda$
7: Compute ciphertext label $L \leftarrow (\mathbf{H}_1(tk'_w))^{K_1}$
8: Encrypt search token $D \leftarrow (\pi(tk_w) \cdot \mathbf{H}_2(tk'_w))^{K_1}$
9: Compute component $C \leftarrow (\pi(op||id))^{K_2} \cdot (\mathbf{G}(tk'_w))^{K_1}$
10: Update private state $\mathbf{State}[w] \leftarrow (tk'_w, cnt_w)$
11: Send the ciphertext (L, D, C) to the server

Server:

12: Store $\mathbf{EDB}[L] \leftarrow (D, C)$

B. The Construction

Algorithms 1, 2, and 3 present the details of Bamboo. To initialize the scheme, protocol Setup takes as inputs a security parameter λ and a maximum padding value $a_{\max}$. Protocol Bamboo.Setup initializes an invertible mapping function, three cryptographic hash functions, and the Diffie-Hellman key exchange protocol parameters. The above-initialized parameters and functions can be publicly known. Then the client initializes a search key K_1, an encryption key K_2, an empty private state $\mathbf{State}$, and an empty database $\mathbf{EDB}$. Finally, it sends $\mathbf{EDB}$ to the server.

DataUpdate. To encrypt an entry $(op, (w, id))$, the client executes protocol DataUpdate. It generates a ciphertext (L, D, C) and sends the generated ciphertext to the server. When generating L, D, and C, the cryptographic hash functions implement the first layer encryption, and the exponentiation operations over group $\mathbb{G}$ implement the second layer encryption. Label L indexes this ciphertext in $\mathbf{EDB}$. Component D encrypts a search token tk_w of the prior ciphertext. The server can use tk_w to find the prior ciphertext and decrypt the search token encrypted in it. Component C encrypts $op||id$.

Search. To perform the secure search for a keyword w, the client executes Search. The client first retrieves the search token tk_w and the DataUpdate counter cnt_w of keyword w from $\mathbf{State}$. If both tk_w and cnt_w are NULL, namely, the client has never issued a DataUpdate query about w, the Search process aborts (Steps 1 and 2). With the search

Algorithm 2 Protocol Bamboo.Search.

Search($K_\Sigma, \mathbf{State}, w; \mathbf{EDB}$)

Client:

1: Retrieve record (tk_w, cnt_w) from $\mathbf{State}[w]$
2: Abort if both tk_w and cnt_w are NULL
3: Establish a temporary secure channel with the server using the Diffie-Hellman key exchange protocol
4: Compute ciphertext label $L \leftarrow (\mathbf{H}_1(tk_w))^{K_1}$
5: Compute $Msk_D \leftarrow (\mathbf{H}_2(tk_w))^{K_1}$ and $Msk_C \leftarrow (\mathbf{G}(tk_w))^{K_1}$
6: Send search trapdoor (K_1, L, Msk_D, Msk_C) to the server via above secure channel

Server:

7: Initialize an empty list $\mathcal{I} \leftarrow \emptyset$
8: Retrieve $(D, C) \leftarrow \mathbf{EDB}[L]$
9: **while** $(D, C) \neq (\text{NULL}, \text{NULL})$ **do**
10: Decrypt search token $tk \leftarrow \pi^{-1}((\frac{D}{Msk_D})^{K_1^{-1}})$
11: Compute $C' \leftarrow \frac{C}{Msk_C}$ and insert C' into $\mathcal{I}$
12: Compute $L \leftarrow (\mathbf{H}_1(tk))^{K_1}$, $Msk_D \leftarrow (\mathbf{H}_2(tk))^{K_1}$, and $Msk_C \leftarrow (\mathbf{G}(tk))^{K_1}$
13: Retrieve $(D, C) \leftarrow \mathbf{EDB}[L]$
14: **end while**
15: Let n be the number of found ciphertexts
16: Pad $\mathcal{I}$ with $a_{\max} - n$ arbitrary elements of $\mathbb{G}$
17: Return $\mathcal{I}$ to the client via the above secure channel

Client:

18: Initialize an empty list $\mathcal{R}$
19: **for** $i = 1$ **to** cnt_w **do**
20: Decrypt the i-th component C'_i of $\mathcal{I}$ by computing $op_i || id_i \leftarrow \pi^{-1}(C'^{K_2^{-1}}_i)$
21: If $op_i = add$, insert id_i into $\mathcal{R}$. Otherwise delete id_i from $\mathcal{R}$.
22: **end for**
23: **return** $\mathcal{R}$

token tk_w, the client computes the label L of the latest generated ciphertext $\mathbf{C} = (L, D, C)$ containing w. Then the client computes Msk_D and Msk_C with tk_w. The server can use Msk_D and Msk_C to decrypt the search token from the component D and partially decrypt the component C of the latest ciphertext (L, D, C) (Steps 4 and 5). Finally, the client sends the search trapdoor (K_1, L, Msk_D, Msk_C) to the server. Upon receiving the search trapdoor, the server uses label L to retrieve the latest issued ciphertext (L, D, C) and uses the search key K_1, Msk_D, and Msk_C to decrypt the search token tk of the prior ciphertext and partially decrypt the component C, respectively. Then, the server uses the decrypted search token tk to locate and decrypt the prior ciphertext. In this way, the server traverses the hidden chain from the latest issued ciphertext of w and finds all the matching ciphertexts (Steps 8 to 14). Then the server pads the size of search results to $a_{\max}$ and returns them to the client. Finally, the client decrypts the first cnt_w ciphertexts with encryption key K_2 and filters out invalid file identifiers according to their operations op.

KeyUpdate. To update the secret key, the client executes KeyUpdate. The client first samples a random element Δ from $\mathbb{Z}_q^*$, and then updates search key K_1 and encryption key K_2 by multiplying those keys by Δ. Then, the client sends the KeyUpdate token Δ to the server. Finally, the server updates the key of the whole encrypted database using Δ.

Algorithm 3 Protocol Bamboo.KeyUpdate.

KeyUpdate($K_\Sigma, \mathbf{State}; \mathbf{EDB}$)

Client:

1: Establish a temporary secure channel with the server using the Diffie-Hellman key exchange protocol
2: Randomly draw the KeyUpdate token Δ from $\mathbb{Z}_q^*$
3: Update local secret keys $K_1 \leftarrow K_1 \cdot \Delta$ and $K_2 \leftarrow K_2 \cdot \Delta$
4: Send Δ to the server via the above secure channel

Server:

5: **for** All (L, D, C) **such that** $(D, C) \leftarrow \mathbf{EDB}[L]$ **do**
6: Update label $L' \leftarrow L^\Delta$
7: Update encrypted search token $D' \leftarrow D^\Delta$
8: Update component $C' \leftarrow C^\Delta$
9: Insert ciphertext $\mathbf{EDB}[L'] \leftarrow (D', C')$
10: Remove ciphertext (L, D, C) from $\mathbf{EDB}$
11: **end for**

Complexity & Cost. Bamboo achieves constant DataUpdate time cost, sub-linear Search complexity, and linear KeyUpdate complexity. The computational complexity of DataUpdate, Search, and KeyUpdate are $O(1)$, $O(a_w)$, and $O(N)$, respectively, where symbol a_w is the total number of DataUpdate queries of searched keyword w, and N is the size of the encrypted database $\mathbf{EDB}$. In terms of bandwidth cost, protocols DataUpdate, Search, and KeyUpdate exchange $O(1)$, $O(a_{\max})$, and $O(1)$ data between the client and the server, respectively. We will evaluate Bamboo's practical efficiency in Section VI. More discussions on Bamboo can be found in Section VII-C.

C. An Example of Bamboo

We give a concrete example for Bamboo in Figure 2. In the beginning, the client holds $K_\Sigma = (K_1, K_2)$ and the private state $(tk_{w,2}, 2)$ of keyword w, where $tk_{w,2}$ is the search token of w's latest generated ciphertext and 2 is the DataUpdate counter value of w. The server stores $\mathbf{C}_1 = (L_1, D_1, C_1)$ and $\mathbf{C}_2 = (L_2, D_2, C_2)$ under w, in which $\mathbf{C}_1$ encrypts $op_1 || id_1$, and $\mathbf{C}_2$ encrypts $op_2 || id_2$ and encapsulates the search token $tk_{w,1}$ of $\mathbf{C}_1$. Next, the thief compromises $K_\Sigma = (K_1, K_2)$ and the private state "$w : (tk_{w,2},\ 2)$". It can accordingly extract the information $(op_1, (w, id_1))$ and $(op_2, (w, id_2))$ from $\mathbf{C}_1$ and $\mathbf{C}_2$, respectively.

Suppose the client does not receive any warnings about the compromise. It continues to run DataUpdate to encrypt $(op_3, (w, id_3))$ with a randomly selected search token $tk_{w,3}$ and K_Σ. The thief knows neither the current information of the private state nor $tk_{w,3}$. Thus, it does not know to which keyword the ciphertext corresponds and cannot extract the information from the newly generated $\mathbf{C}_3 = (L_3, D_3, C_3)$. This ciphertext leaks nothing to the thief. At last, the client updates its key K_Σ from (K_1, K_2) to (K'_1, K'_2). The thief cannot use (K_1, K_2) to decrypt any ciphertexts under (K'_1, K'_2).

After the KeyUpdate, the client can use K'_1 and $tk_{w,3}$ to generate the trapdoor for a new search query. The server first locates and decrypts $\mathbf{C}_3$ to obtain $tk_{w,2}$. It further uses K'_1 and $tk_{w,2}$ to identify and decrypt $\mathbf{C}_2$ to get $tk_{w,1}$.

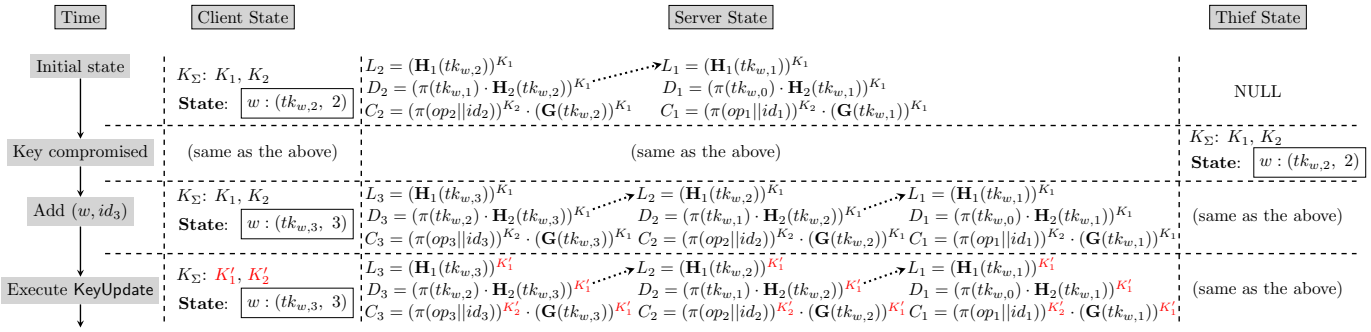


Fig. 2. An example of Bamboo. In the beginning, the client has run DataUpdate with $(op_1, (w, id_1))$ and $(op_2, (w, id_2))$. After the compromise, if the client is not "warned" immediately, it may still perform a new DataUpdate query on $(op_3, (w, id_3))$. At last, the client executes KeyUpdate for K_Σ.

Similarly, the server traverses back to the first ciphertext $\mathbf{C}_1$ (along the chain). It eventually returns the partially decrypted components C_1, C_2, and C_3 to the client who will fully recover $(op_1, (w, id_1))$, $(op_2, (w, id_2))$, and $(op_3, (w, id_3))$.

D. Correctness and Security Analysis

Correctness. The correctness of Bamboo comes from the collision-resistance of hash functions $\mathbf{H}_1$, $\mathbf{H}_2$, and $\mathbf{G}$, the algebraic features of group $\mathbb{G}$, and the correctness of the invertible mapping function π. Specifically, when executing DataUpdate with a given entry $(op, (w, id))$, both the collision-resistance of $\mathbf{G}$ and $\mathbf{H}_2$ and the correctness of π guarantee that the generated ciphertext (L, D, C) encrypts op and id in component C and encrypts the previously issued ciphertext's search token tk_w in component D correctly.

When executing Search with a given keyword w, the latest issued ciphertext (L, D, C) can be correctly located by the label $L = (\mathbf{H}_1(tk_w))^{K_1}$ in the search trapdoor. Given K_1, Msk_D, and Msk_C contained in the search trapdoor, the server can decrypt the search token tk of the prior ciphertext from D and partially decrypt component C to get $(\pi(op||id))^{K_2}$. Then, the server can use search token tk to locate and decrypt the prior ciphertext with hash functions $\mathbf{H}_1$, $\mathbf{H}_2$, and $\mathbf{G}$. In this way, the server can precisely find all matching ciphertexts and return the correctly partially decrypted ciphertexts to the client. Finally, the client can use the encryption key K_2 to decrypt returned results.

Without loss of generality, suppose there exists a ciphertext $(L, D, C) = (L_0^{K_1}, D_0^{K_1}, C_0^{K_2} \cdot X_0^{K_1})$, where K_1 is the search key, and K_2 is the encryption key. After executing protocol KeyUpdate with a KeyUpdate token Δ, the post-key-updated ciphertext is $(L_0^{K_1 \cdot \Delta}, D_0^{K_1 \cdot \Delta}, C_0^{K_2 \cdot \Delta} \cdot X_0^{K_1 \cdot \Delta})$, and the two new keys are $K_1 \cdot \Delta$ and $K_2 \cdot \Delta$. Let $K'_1 = K_1 \cdot \Delta$ and $K'_2 = K_2 \cdot \Delta$. It is clear that the new keys K'_1 and K'_2 can be used to search and decrypt the post-key-updated ciphertext.

Security. The post-compromise security of Bamboo against Type 1 model is captured according to the views of the server $\mathcal{A}_{\text{Srv}}$ and the thief $\mathcal{A}_{\text{Thf}}$. For $\mathcal{A}_{\text{Srv}}$, protocol Setup leaks only the security parameter λ and the maximum padding value $a_{\max}$; protocol DataUpdate leaks nothing; when issuing a search query of keyword w, protocol Search leaks search pattern $\mathsf{sp}(w)$, file identifiers matching w and the insertion timestamps of those file identifiers (i.e., $\mathsf{TimeDB}(w)$), the DataUpdate timestamps of w (i.e., $\mathsf{DUTime}(w)$), and the KeyUpdate histories $\mathsf{KUHist}(U_{\text{now}})$; protocol KeyUpdate leaks nothing but $\mathsf{KUHist}(U_{\text{now}})$. The formal definitions of leakage functions $\mathsf{sp}(w)$, $\mathsf{TimeDB}(w)$, and $\mathsf{DUTime}(w)$ are described as:

$$\begin{aligned}\mathsf{TimeDB}(w) = \{(u, id) \mid (u, add, (w, id)) \in Q^{\text{DU}} \text{ and } \\ \forall u', (u', del, (w, id)) \notin Q^{\text{DU}}\},\\ \mathsf{sp}(w) = \{u \mid (u, w) \in Q^{\text{Srch}}\},\\ \mathsf{DUTime}(w) = \{u \mid (u, op, (w, id)) \in Q^{\text{DU}}\}.\end{aligned}$$

As for $\mathcal{A}_{\text{Thf}}$, protocol Setup leaks the security parameter λ and the maximum padding value $a_{\max}$; protocols DataUpdate, Search, and KeyUpdate leak nothing; when the secret key is compromised, the thief $\mathcal{A}_{\text{Thf}}$ learns the plaintexts of the ciphertexts that were generated by the last execution of protocol KeyUpdate and the DataUpdate queries issued since the last execution of KeyUpdate.

Formally, we have the following Theorem 1, whose proof can be found in the full version of this paper. Moreover, the leakage functions of Bamboo in the view of the server satisfy the forward security and the backward security in the sense that protocol Search only leaks *the file-identifiers currently matching the queried keyword w, when they were uploaded, and when all the DataUpdate on w happened* [12] to the server.

Theorem 1. *Suppose hash functions $\mathbf{H}_1$, $\mathbf{H}_2$, and $\mathbf{G}$ are random oracles, and DDH assumption holds in $\mathbb{G}$, Bamboo is a post-compromise-secure SEKU scheme since:*

1) *Bamboo is $\mathcal{L}_{Srv}$-adaptively secure and the leakage functions $\mathcal{L}_{Srv} = (\mathcal{L}_{Srv}^{Stp}, \mathcal{L}_{Srv}^{DaUpdt}, \mathcal{L}_{Srv}^{Srch}, \mathcal{L}_{Srv}^{KeyUpdt})$ can be written as:*

$$\begin{aligned}\mathcal{L}_{Srv}^{Stp}(\lambda, a_{max}) = (\lambda, a_{max}),\\ \mathcal{L}_{Srv}^{DaUpdt}(op, (w, id)) = NULL,\\ \mathcal{L}_{Srv}^{KeyUpdt} = \mathcal{L}'_{Srv}(\mathit{KUHist}(U_{now})),\\ \mathcal{L}_{Srv}^{Srch}(w) = \mathcal{L}''_{Srv}(\mathsf{sp}(w), \mathit{TimeDB}(w),\\ \mathit{DUTime}(w), \mathit{KUHist}(U_{now})),\end{aligned}$$

   where $\mathcal{L}'_{Srv}$ and $\mathcal{L}''_{Srv}$ are two stateless functions.

2) *Bamboo is $\mathcal{L}_{Thf}$-adaptively secure and the leakage functions $\mathcal{L}_{Thf} = (\mathcal{L}_{Thf}^{Stp}, \mathcal{L}_{Thf}^{DaUpdt}, \mathcal{L}_{Thf}^{Srch},$*

$\mathcal{L}_{Thf}^{KeyUpdt}, \mathcal{L}_{Thf}^{KeyLeak}$) *can be written as:*

$$\mathcal{L}_{Thf}^{Stp}(\lambda, a_{max}) = (\lambda, a_{max}),$$
$$\mathcal{L}_{Thf}^{DaUpdt}(op, (w, id)) = NULL,$$
$$\mathcal{L}_{Thf}^{Srch}(w) = NULL,\ \mathcal{L}_{Thf}^{KeyUpdt} = NULL,$$
$$\mathcal{L}_{Thf}^{KeyLeak} = \mathcal{L}'_{Thf}(\mathit{CUHist}(U_{now}), \mathit{DUHist}(U_{now})),$$

where $\mathcal{L}'_{Thf}$ is a stateless function.

E. Improvements on Bandwidth

In Section IV-C, we define leakage functions $\mathcal{L}_{\text{Thf}}^{Srch}$ and $\mathcal{L}_{\text{Thf}}^{KeyLeak}$ for the thief to be stateless. To achieve this stateless property, a post-compromise-secure SEKU instance has to strictly protect the result volume of each search query from $\mathcal{A}_{\text{Thf}}$. Thus, Bamboo pads the size of search results to a predefined maximum padding value a_{max}. Moreover, this padding clearly causes extra bandwidth.

To reduce this cost while protecting search queries, we propose to apply a flexible padding technique to protocol Search. We consider the adjustable padding technique introduced by Demertzis et al. [27].

Adjustable Padding Overview. Demertzis et al. [27] applied this technique to build a secure static searchable encryption scheme. Specifically, given a parameter $x(x \geq 2)$ and a static database $\mathbf{DB}$, let $\mathbf{DB}(w)$ be the set of corresponding file identifiers to keyword w. When encrypting $\mathbf{DB}(w)$ for a keyword w in the Setup process, the client finds an integer i such that $x^{i-1} < |\mathbf{DB}(w)| \leq x^i$ and pad $x^i - |\mathbf{DB}(w)|$ dummy entries to the encrypted results of $\mathbf{DB}(w)$. The adjustable padding technique guarantees that the search on a keyword w only leaks to the server the result volume of size $\log_x|\mathbf{DB}(w)|+1$, namely, leaking only $\log_2\log_x|\mathbf{DB}(w)|+1$ bits information.

Unfortunately, we cannot apply the above adjustable padding technique directly to Bamboo since it may lead to a severe security problem. For example, suppose $\mathcal{A}_{\text{Thf}}$ compromises the secret key, and there is a keyword w' with a unique adjustable padding value pad_{uni}. In this case, when observing that the search results of a client's Search query has the size of pad_{uni}, the thief $\mathcal{A}_{\text{Thf}}$ may have a high probability of determining that the client is searching for w'.

Algorithm 4 Function PaddingVal-adj$(a_{\text{max}}, x, w, \mathbf{State})$.

1: Retrieve $(tk_w, cnt_w) \leftarrow \mathbf{State}[w]$
2: Find an integer i such that $x^{i-1} < cnt_w \leq x^i$
3: If there is not a second keyword w' of which DataUpdate counter $cnt_{w'}$ satisfies $x^{i-1} < cnt_{w'} \leq x^i$, return a_{max}
4: If $x^i > a_{\text{max}}$, return a_{max}
5: Randomly return a_{max} or x^i

To tackle this problem, we propose a new padding method named PaddingVal-adj (see Algorithm 4). It takes the maximum padding value a_{max}, an integer x $(x \geq 2)$, a keyword w, and private state $\mathbf{State}$ as inputs, and calculates the adjustable padding value x^i for keyword w according to DataUpdate counter cnt_w. Next, it checks if x^i can be used to *deterministically* distinguish w from other keywords and returns a_{max} if so. Otherwise, the function returns a_{max} if $x^i > a_{\text{max}}$. Finally, the function randomly chooses and returns one of x^i and a_{max} as the padding value. PaddingVal-adj hides more information than the adjustable padding technique from the thief $\mathcal{A}_{\text{Thf}}$ who may have (prior) plaintext knowledge about the encrypted database. Besides, by applying the function, a Search query can take less bandwidth to be completed than using the maximum padding technique.

Note that Step 3 (in Algorithm 4) yields the Search leakage to the server. Specifically, this step determines if the function returns the maximum padding value a_{max} based on the keyword frequency of w. On the other hand, the frequency could possibly be leaked to the server through the padding value. The "random return" strategy in Step 5 (Algorithm 4) is to reduce the above leakage by weakening the server's ability to check whether a_{max} is selected randomly or computed from the keyword frequency.

For example, suppose the client's encrypted database only contains w_1 and w_2, and the (search) response sizes for both keywords are $x^{i'}$ for certain x and i'. After the client issues Search queries on w_1 and w_2, the server can extract from the Search leakage that: (1) the client only uses two distinct keywords in the database and (2) the keywords have the same response size of $x^{i'}$. Note the server here cannot directly see w_1 and w_2. In this context, we explain what will happen if Step 5 deterministically returns the adjustable value. Upon searching for w_1 and w_2, the client should require the server to pad the responses to $x^{i'}$. If the client issues a DataUpdate query and then a Search query on w_1 (or w_2), the server can easily learn the information about the DataUpdate query. More concretely, if the Search query requires the server to pad the results to $x^{i'}$, the server will know the DataUpdate query contains a keyword different from w_1 and w_2. This is because the condition of Step 3 does not hold. Otherwise (i.e., if the requested padding size is a_{max}), the server learns that two queries contain distinct keywords, for example, the DataUpdate query contains w_1 while w_2 is in the Search query, or the other way round. We note that the "random" strategy helps us to hide the above connections between keywords and queries from the server.

One can easily apply function PaddingVal-adj in protocol Search of Bamboo to reduce the bandwidth. Specifically, in protocol Setup, the client takes an integer $x(x \geq 2)$ as an additional input. In protocol Search, before sending a search trapdoor to the server, the client executes function PaddingVal-adj with the maximum padding value a_{max}, integer x, the queried keyword w, and the private state $\mathbf{State}$ as inputs to compute the padding value num_{pad}. Then, the client sends num_{pad} along with the search trapdoor to the server. After finding all matching ciphertexts, the server pads the size of the search results to num_{pad} and returns the padded search results to the client. For convenience, we name the resulting scheme Bamboo*. Section VI will experimentally test and compare the Search performance of Bamboo* and Bamboo.

Compared with Bamboo, Bamboo* leaks more information to both $\mathcal{A}_{\text{Srv}}$ and $\mathcal{A}_{\text{Thf}}$. For $\mathcal{A}_{\text{Srv}}$, the padding value during a search should be included in the Search leakage function. In practice, Bamboo* is feasible to handle large-scale databases, such as Wikipedia. When doing so, it is hard for $\mathcal{A}_{\text{Srv}}$ to infer information from unsearched ciphertexts with padding

values. In this case, the leakage function $\mathcal{L}_{\text{Srv}}^{Srch}$ is still stateless. Namely, under this assumption, protocol $\texttt{Bamboo}^*$.Search has the leakage function

$$\mathcal{L}_{\text{Srv}}^{Srch}(w) = \mathcal{L}''_{\text{Srv}}(\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{DUTime}(w), \mathsf{KUHist}(U_{\text{now}}), num_{\text{pad}}),$$

where $\mathcal{L}''_{\text{Srv}}$ is a stateless function.

For $\mathcal{A}_{\text{Thf}}$, the leakage functions of protocol $\texttt{Bamboo}^*$.Search and the key-compromise event KeyLeak can no longer be stateless. If the search result volume of a keyword is an adjustable padding value x^i, the thief $\mathcal{A}_{\text{Thf}}$ can gain a higher probability of guessing - what the client is searching for - than the case where the search result volume always equals $a_{\max}$. Thus, in $\texttt{Bamboo}^*$, the leakage functions $\mathcal{L}_{\text{Thf}}^{Srch}$ and $\mathcal{L}_{\text{Thf}}^{KeyLeak}$ are described as:

$$\mathcal{L}_{\text{Thf}}^{Srch}(w) = \overline{\mathcal{L}}_{\text{Thf}}(num_{\text{pad}}),$$
$$\mathcal{L}_{\text{Thf}}^{KeyLeak} = \overline{\overline{\mathcal{L}}}_{\text{Thf}}(\mathsf{CUHist}(U_{\text{now}}), \mathsf{DUHist}(U_{\text{now}})),$$

where $\overline{\mathcal{L}}_{\text{Thf}}$ and $\overline{\overline{\mathcal{L}}}_{\text{Thf}}$ are two stateful functions.

VI. Implementations and Evaluations

We implemented Bamboo, $\texttt{Bamboo}^*$, and three key-updatable DSSE schemes and further compared their performance using a real-world dataset. Those three baseline schemes are revised from DSSE schemes that have the same level of backward security with Bamboo. Specifically, the selected schemes are MITRA [18], Fides [12], and Aura [65]. We denote their corresponding key-updatable versions by $\texttt{MITRA}^{\text{KU}}$, $\texttt{Fides}^{\text{KU}}$, and $\texttt{Aura}^{\text{KU}}$, respectively. We did not choose $\texttt{SD}_a$ and $\texttt{SD}_d$ for comparison because Sun et al. have proved that Aura outperforms $\texttt{SD}_a$ and $\texttt{SD}_d$ [65]. The KeyUpdate processes of $\texttt{Fides}^{\text{KU}}$ and $\texttt{Aura}^{\text{KU}}$ are interactive. Namely, the client needs to download, decrypt and re-encrypt, and re-upload the entire encrypted database to update the secret key. $\texttt{MITRA}^{\text{KU}}$ was implemented by replacing the PRF function of MITRA with a key-updatable one and encapsulating the file identifier with the mapping function π. In this way, $\texttt{MITRA}^{\text{KU}}$ is equipped with a non-interactive KeyUpdate protocol. As discussed in Section I, $\texttt{Aura}^{\text{KU}}$, $\texttt{Fides}^{\text{KU}}$, and $\texttt{MITRA}^{\text{KU}}$ cannot achieve post-compromise security, since they cannot guarantee the ciphertext security generated during the special time slot. Note that we did not implement a padding process during the search for the baseline key-updatable DSSE schemes.

A. Experimental Setup

We used a client and a server connected via a LAN network to perform the experiments. Table II presents the hardware and operating system configurations of the client and the server, respectively. They are connected via the Ethernet with about 100 Mbps bandwidth and about a one-millisecond delay. To yield comprehensive experiments, we additionally created a network environment with about 300 milliseconds delay. The extra network delay is produced with the "tc" command offered by the operating system.

We coded Bamboo, $\texttt{Bamboo}^*$, and the baseline schemes in C++. All the evaluated schemes use a native TCP socket to establish network communications, SQLite database [23] as their client states, and PostgreSQL database [35] to store ciphertexts. In particular, we used the command "PRAGMA synchronous=off" to disable the database synchronization mechanism of SQLite. Hash functions, PRF functions, and the invertible mapping functions in Bamboo, $\texttt{Bamboo}^*$, and $\texttt{MITRA}^{\text{KU}}$ are implemented with OpenSSL [30] (which provides SHA-256, SHA-384, and SHA-512 cryptographic hash functions) and Relic Toolkit [2] (which provides NIST-P256 elliptic curve algorithms). We used the GMP library [29] to realize the RSA-based permutation that is used in $\texttt{Fides}^{\text{KU}}$. $\texttt{Aura}^{\text{KU}}$ is developed based on the code provided by Sun et al. All implementations can achieve a 128-bit security level.

TABLE II. Hardware and OS Configurations.

	Client	Server
CPU	AMD Ryzen 9 5950X	Intel Xeon Silver 4216
Memory	128 GB	128 GB
Disk Drive	256 GB SAMSUNG PM981 NVME SSD	
OS	Ubuntu Server 20.04 x64	

Our test dataset is extracted from English Wikipedia [31]. Specifically, we used WikiExtractor [3] to process the Wikipedia [31] and then ran porter stemmer [61] to extract keywords from the processed data. We treated one article as a single document and directly used the identifier number of each article as the file identifier. The length of the file identifier is, at most, 8 bytes. We also chose some of the extracted data to produce a dataset containing 3,257,613 pairs of keyword and file identifiers. The dataset contains twenty-five keywords. The number of those twenty-five keywords matching file identifiers ranges from about 10,000 to about 250,000. Those twenty-five keywords are sufficient to yield a comprehensive evaluation of time costs. To match with the dataset, we adaptively fine-tuned the parameter of $\texttt{Aura}^{\text{KU}}$, especially the supported maximum number of deletions $d = 150,000$, the false positive rate $p = 10^{-5}$, and the hash function number $h = 13$ of the underlying Bloom Filter. We also set the parameter $a_{\max} = 410,000$ of Bamboo and $\texttt{Bamboo}^*$. In $\texttt{Bamboo}^*$, we set the integer $x = 2$, which will be used in function PaddingVal-adj. Note $\texttt{MITRA}^{\text{KU}}$ and $\texttt{Fides}^{\text{KU}}$ do not have such parameters for fine-tuning.

In the experiments, we comprehensively tested and compared the performance of DataUpdate, KeyUpdate, and Search of Bamboo, $\texttt{MITRA}^{\text{KU}}$, $\texttt{Fides}^{\text{KU}}$, and $\texttt{Aura}^{\text{KU}}$ using our dataset. Since the essential difference between Bamboo and $\texttt{Bamboo}^*$ is in the Search protocol, we only evaluated the Search performances of them. Specifically, when testing DataUpdate performance, we reported those schemes' average client time costs to generate one add DataUpdate query when encrypting the entire database. We also tested the average client time costs on generating one del DataUpdate query when issuing $d = 150,000$ del queries. In addition, we evaluated the client time costs on reading and then writing the client state when generating a DataUpdate query with $op = del$ or $op = add$.

The KeyUpdate and Search performance are tested in both network environments with delays of about one millisecond and 300 milliseconds, respectively. While testing the KeyUpdate performance, we encrypted the whole dataset

https://github.com/MonashCybersecurityLab/Aura

and then updated the encryption key of the generated ciphertexts to evaluate the total time and client time costs for the schemes. We evaluated the Search performance under two conditions - with and without (document) deletion. The evaluation metrics are the same as those we used in the experiments for KeyUpdate. In the examination of the Search performance without deletion, we encrypted the dataset and then performed searches for all keywords. The experiment for the case with deletion is similar, but with the exception that after encrypting the dataset, we chose a keyword with about 150,000 matching file identifiers and further issued different numbers of DataUpdate queries with $op = del$ of the keyword to the server. These queries are made up of 0%-90% randomly selected file identifiers of the corresponding search results to the keyword. We then evaluated the Search efficiency on the keyword.

Finally, we compared the client and server storage costs of Bamboo, MITRA^{KU}, Fides^{KU}, and Aura^{KU} (note Bamboo and Bamboo* share the same client/server storage complexity). We extracted additional keywords from English Wikipedia and run DataUpdate with them to test how client-side storage grows with the number of distinct keywords. We did not report the impacts of file numbers on the client storage as, in the implementations, all counters related to files were fixed to 4-byte integers. In terms of the server-side storage, we reported the size of PostgreSQL tables under various numbers of add entries (i.e., the entries having $op = add$).

B. DataUpdate Performance

TABLE III. COMPARISON ON AVERAGE CLIENT TIME COST (μS) OF DATAUPDATE.

	Bamboo	MITRA^{KU}	Aura^{KU}	Fides^{KU}
$op = add$	1.96×10^3	1.08×10^3	7.07×10^4	6.19×10^3
$op = del$	1.96×10^3	1.08×10^3	7.04×10^4	6.14×10^3
Client State	59.73	54.18	6.97×10^4	55.21

We present the evaluation of DataUpdate in Table III. Note that the average DataUpdate time costs of all the compared schemes are constant, and they are not affected by any historical DataUpdate, KeyUpdate, and Search queries. We can see that Bamboo outperforms Aura^{KU} and Fides^{KU}. Specifically, Bamboo saves about 97.22% and 68.33% client time costs compared to Aura^{KU} and Fides^{KU}. Table III also presents the time costs of accessing the SQLite-based client state when issuing a DataUpdate query. The client state cost is the same in both cases of $op = add$ and $op = del$. Over 98% of Aura^{KU}'s DataUpdate overhead comes from accessing the client state. This may indicate the importance of maintaining lightweight access (of client state) in practice. Fortunately, Bamboo satisfies this requirement. Say, it only consumes about 59.73 microseconds to access the SQLite database, and the result is very close to the cost of MITRA^{KU} and Fides^{KU} (in which the performance gap is < 6 microseconds). Compared to MITRA^{KU}, Bamboo only requires a slight extra cost ($\leq 8.80\times10^2$ microseconds) to issue a DataUpdate query. This is because Bamboo produces one more part in the ciphertext of a single entry (than MITRA^{KU}) to maintain the chain-link inter-ciphertext structure, which results in an extra exponentiation operation over the elliptic curve element.

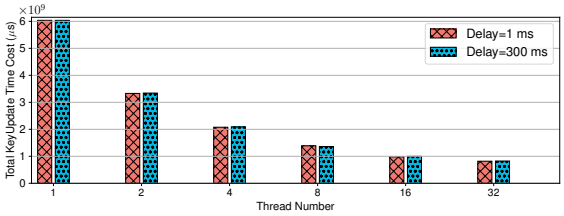


Fig. 3. Total KeyUpdate Time Cost of Bamboo vs. Number of Threads.

C. KeyUpdate Performance

TABLE IV. COMPARISON ON KEYUPDATE PERFORMANCE (μS).

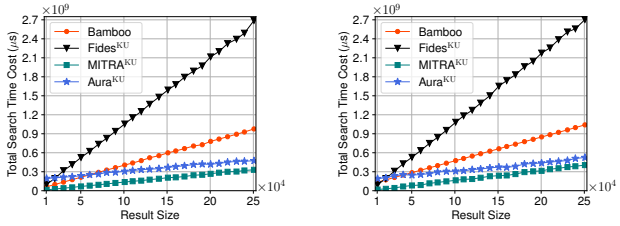
Delay (ms)		Bamboo	MITRA^{KU}	Aura^{KU}	Fides^{KU}
1	Total Time Cost	6.04×10^9	4.15×10^9	9.06×10^9	2.07×10^{10}
	Client Time Cost	24.49	12.93	1.34×10^9	2.02×10^{10}
300	Total Time Cost	6.04×10^9	4.15×10^9	1.55×10^{10}	2.21×10^{10}
	Client Time Cost	19.20	10.90	1.36×10^9	2.02×10^{10}

We show the comparisons on KeyUpdate in Table IV. Bamboo outperforms Aura^{KU} and Fides^{KU} in all the metrics. Specifically, when the network delay is 300 milliseconds, the total time cost of Bamboo to update the key of the encrypted database is only approximately 6.04×10^9 microseconds (about 100 minutes), which is about 3.66 times and 2.57 times faster than those of Fides^{KU} and Aura^{KU}, respectively. The client time cost of Bamboo is almost negligible compared to the overheads brought by Aura^{KU} and Fides^{KU}. We notice that the network quality makes less impact on Bamboo than Aura^{KU} and Fides^{KU}. For example, the absolute difference of Bamboo's KeyUpdate time costs between the two network environments is about 6.77×10^5 microseconds (0.67 seconds), while Aura and Fides incur approximately 1.01×10^9 and 1.41×10^8 microseconds, respectively. This is because Bamboo's client only transfers one small token to the server to execute KeyUpdate while the clients of Aura^{KU} and Fides^{KU} have to download and re-upload the whole encrypted database. As the increase of network delay, Bamboo may provide more advantage in KeyUpdate than Aura^{KU} and Fides^{KU}. Bamboo takes longer than MITRA^{KU} in KeyUpdate, because it maintains one more component in a single ciphertext (than MITRA^{KU}) to achieve both post-compromise security and sub-linear (search) complexity.

The KeyUpdate protocol of Bamboo can be accelerated with multi-thread technique in practice. Figure 3 reports the total time cost of Bamboo's KeyUpdate when running with different numbers of threads in two network environments. The results indicate that using multi-thread can help us to reduce the time cost, and they further confirm again that the network quality makes little impact on the KeyUpdate performance. For example, leveraging sixteen threads, the KeyUpdate only takes nearly 9.80×10^8 microseconds (about 16.33 minutes) to update the key of the whole database in both network environments, which is approximately 6.16 times faster than the single-thread approach. With this trend, updating the whole database can lead to less cost as the increase of thread number.

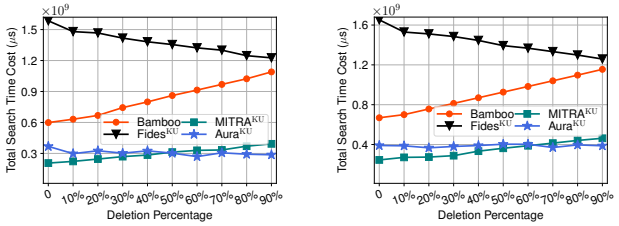
D. Search Performance

Search without Deletion. Figure 4 reports the total search time costs of the evaluated schemes. The figure clearly shows



(a) Delay=1 ms. (b) Delay=300 ms.

Fig. 4. Total Search Time Cost vs. Result Size without Deletion.



(a) Delay=1 ms. (b) Delay=300 ms.

Fig. 6. Total Search Time Cost vs. Percentage of Deletion.

the linear relationships between the Search time costs and result sizes of all the compared schemes. We see that when the result size $> 20,000$, Bamboo outperforms $\texttt{Fides}^{\text{KU}}$ in both network environments. We state that when the result size $\leq 20,000$, the performance gap between Bamboo and $\texttt{Fides}^{\text{KU}}$ is not significant. For example, when the result size is approximately $10,000$ in both network environments, Bamboo costs at most extra 4.20×10^7 microseconds (about 0.70 minutes) than $\texttt{Fides}^{\text{KU}}$. The results show that the network delay makes less impact on Bamboo than $\texttt{MITRA}^{\text{KU}}$ in the stage of Search. For example, searching for the keyword corresponding to about $250,000$ matching ciphertexts, the Bamboo's performance with 300 ms delay is 6.57% worse than that of the case when Delay=1 ms; and similarly, $\texttt{MITRA}^{\text{KU}}$ consumes about 22.46% more cost with 300 ms delay. We also see that Bamboo is less efficient than $\texttt{Aura}^{\text{KU}}$. However, this gap is acceptable. For example, in both network environments, Bamboo just requires at most an extra 1.82×10^3 microseconds than $\texttt{Aura}^{\text{KU}}$ to find a matching ciphertext, on average. Bamboo requires more exponentiation operations over the elliptic curve elements than $\texttt{MITRA}^{\text{KU}}$ and $\texttt{Aura}^{\text{KU}}$ during the search. Those operations, despite their high computational costs, are necessary for the server to search over the chain-like structure correctly.

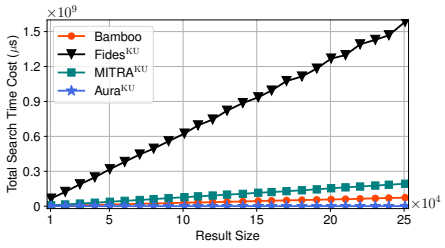


Fig. 5. Client Search Time Cost vs. Result Size without Deletion.

Figure 5 presents the experimental results about client time costs on Search. Since the network delay (no matter how long the delay is) does not affect the cost of the client side, we only present the results when Delay= 1 ms. In this experiment, one may see that Bamboo performs better than both $\texttt{Fides}^{\text{KU}}$ and $\texttt{MITRA}^{\text{KU}}$. Specifically, when the result size is about 250,000, the cost of Bamboo is only around 7.42×10^7, saving at least 95.32% and 61.65% overheads as compared to $\texttt{Fides}^{\text{KU}}$ and $\texttt{MITRA}^{\text{KU}}$, respectively. If the result size continues increasing, Bamboo will reduce more time compared to $\texttt{Fides}^{\text{KU}}$ and $\texttt{MITRA}^{\text{KU}}$. We notice $\texttt{Aura}^{\text{KU}}$ has a small advantage over Bamboo, as the Bamboo's client has to perform decryption to obtain the final results.

Search with Deletion. Figure 6 shows the total search time cost as the deletion percentage varies in the network environments. From the results, we can conclude that network delay does not significantly affect the Search performance of Bamboo when there are historical deletion queries. Specifically, Bamboo has an overhead of 8.54×10^7 microseconds (1.42 minutes) to complete the Search in Figure 6(b) as compared to Figure 6(a) w.r.t. the same keyword. With the increase of the deletion percentage, the cost of $\texttt{Fides}^{\text{KU}}$ decreases. The reason is that $\texttt{Fides}^{\text{KU}}$'s client needs to re-encrypt and re-upload fewer ciphertexts during Search (as the deletion number increases). Even so, in general, Bamboo still outperforms $\texttt{Fides}^{\text{KU}}$. For example, when the percentage is set to 40% in Figure 6(b), Bamboo consumes around 8.70×10^8 microseconds in total, which reduces 39.84% cost compared with $\texttt{Fides}^{\text{KU}}$. Even in the worst case, say, the percentage=90% in Figure 6(b), Bamboo costs at most another 5.09×10^4 microseconds, compared with others, to find a matching ciphertext. This performance gap exists due to the same reason explained in the "search without deletion".

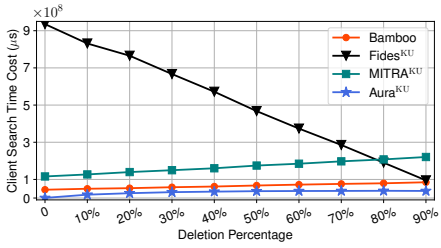
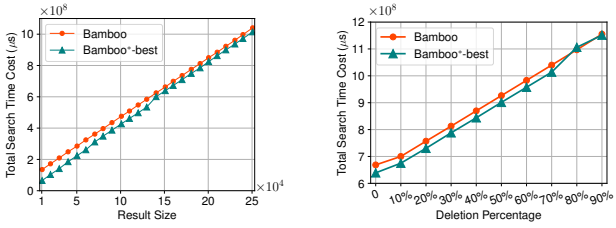


Fig. 7. Client Time Cost vs. Percentage of Deletion.

Figure 7 reports the cost on the client side with the change in deletion percentage. Since the network delay does not affect the client cost, we only present the results when Delay=1 ms. Bamboo keeps its advantages over $\texttt{Fides}^{\text{KU}}$ and $\texttt{MITRA}^{\text{KU}}$. This indicates that Bamboo's Search is cost-effective and computation friendly to the client in general. When the percentage is 40% in Figure 7, our client only takes about 6.19×10^7 microseconds (roughly 1.03 minutes) in Search, approximately 9.24 times and 2.59 times more efficient than $\texttt{Fides}^{\text{KU}}$ and $\texttt{MITRA}^{\text{KU}}$, respectively. One may also see that the gap between Bamboo and $\texttt{Aura}^{\text{KU}}$ is actually quite close. For example, when the percentage=90% Bamboo take 3.09×10^3 microseconds more to locate a matching ciphertext than $\texttt{Aura}^{\text{KU}}$. This is because Bamboo requires the client to perform decryption and filter out the deleted file identifiers to obtain the search results, while $\texttt{Aura}^{\text{KU}}$ does not.



(a) Total Search Time Cost vs. Result Size without Deletion.

(b) Total Search Time Cost vs. Percentage of Deletion.

Fig. 8. Search Comparison Bamboo vs. Bamboo*.

Search Comparison between Bamboo and Bamboo*. In this part, we compare the Search Performance with and without historical deletions between Bamboo and Bamboo*. For the case when Delay=1 ms, the performance gap between the two schemes is not significant. Therefore, we only present the experimental results when Delay=300 ms. For Bamboo*, we tested and recorded the best case of its Search. This case happens when the function PaddingVal-adj returns the adjustable padding value (instead of the maximum padding one) if the padding value could be used. We note that the worst case is the other way round, i.e., returning the maximum padding value, which is equal to the Bamboo's Search. We name the best case Bamboo*-best. From Figure 8, we see that the smaller the number of matching ciphertexts we have, the more time, in the Bamboo*-best, we save as compared to Bamboo. We also notice that in both sub-figures, the gap of both lines is shrinking as the increase of result size and deletion percentage. For instance, in Figure 8(a), when the result size is about 10,000, Bamboo*-best saves about 70.32 seconds over Bamboo to complete the Search; whilst the size reaches about 250,000, the advantage decreases to 25.67 seconds. Similarly, assuming the deletion percentage=30% (namely, there are in total 105,697 matching ciphertexts), the cost we save is about 24.80 seconds in Figure 8(b). Then, if the percentage is greater than 80%, the gap disappears. This is because when the percentage is greater than 80%, the adjustable padding value is greater than the maximum padding value, and thus the maximum padding value is used to complete the padding process.

E. Storage Efficiency

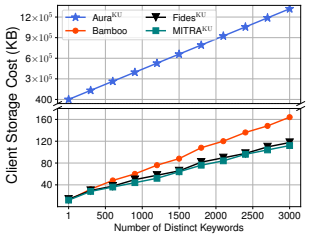


Fig. 9. Client Storage Cost vs. Number of Distinct Keywords

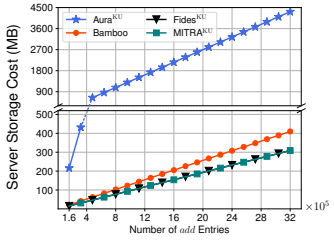


Fig. 10. Server Storage Cost vs. Number of *add* Entries

Figure 9 presents a comparison of the client-side storage costs. Bamboo clearly outperforms Aura^{KU}. When the client executes DataUpdate with 2,100 distinct keywords, Bamboo only requires roughly 120.06 KB data, saving 99.98% storage as compared to Aura^{KU}. This is because Bamboo only records a $\lambda-$bit random number in the private state and a 4-byte counter for each keyword, but Aura^{KU} applies a Bloom Filter (with tens of thousands of bits). The Bamboo's client stores more data than MITRA^{KU} and Fides^{KU}. That extra overhead is acceptable in practice. For example, given 3,000 distinct keywords, Bamboo takes about 164.06 KB, in which there are only $\leq$ 52.03 KB extra costs as compared to MITRA^{KU} and Fides^{KU}.

Figure 10 shows the server storage costs under various numbers of *add* entries. We see that Bamboo surpasses Aura^{KU}. For instance, given 2,400,000 *add* entries, the Bamboo's server consumes 308 MB storage, which is about 9.5% of the cost of Aura^{KU} (3,242 MB). Compared to MITRA^{KU} and Fides^{KU}, the Bamboo's ciphertext of a single (*add*) entry yields an extra part. This additional cost is minor. Assuming the encrypted database contains 3,200,000 entries, the cost is only 101 MB on the server side.

As a conclusion, considering the total cost in KeyUpdate and DataUpdate, Bamboo outperforms Fides^{KU} and Aura^{KU} while achieving similar performance to MITRA^{KU}. It is interesting to see that KeyUpdate of Bamboo can be accelerated via the multi-thread technique. In terms of Search efficiency, Bamboo maintains the same level of practicability as those baseline schemes. In addition, Bamboo gains noticeable advantages on the client side among those schemes in terms of the costs in DataUpdate, KeyUpdate, and Search. Moreover, Bamboo achieves practical storage performance, which is very close to that of MITRA^{KU} and Fides^{KU}, on both the client and server sides. Given that Bamboo captures the post-compromise security, we can conclude that it is the first practical DSSE scheme with high performance and strong security in the literature. As an improved variant, Bamboo* can save Search time cost. Moreover, it is applicable to those large-scale databases where there exist many keywords which correspond to a small amount of matching file identifiers, such as the English Wikipedia. We provide more discussions on the above experiments in Section VII-D.

VII. Discussions and Future Works

A. Discussions on Trivial Extension to SEKU

Some of the existing DSSE schemes (e.g., MITRA, ORION, HORUS [18]) can be extended to support KeyUpdate by replacing their PRF functions or symmetric encryption schemes with key-updatable PRF [8], [71] or updatable encryption [13], [41], [47], [50]. However, such an extension may not achieve post-compromise security. As we have pointed out in Section I, they cannot guarantee the security of ciphertexts generated in the special time slot (i.e., after the key compromise before the KeyUpdate). This is because they cannot provide enough *unpredictable private randomness* to generate a ciphertext. Private randomness guarantees that the randomnesses (e.g., the honest, randomly generated secret key) should be only known to the client. In each of the existing schemes, the DataUpdate protocol generates one or more parts of the ciphertext with only static or derivable private randomness, e.g., a secret key of a CPA-secure encryption scheme or a secret key derived from a counter. Once the secret key and the private state are exposed, the thief can easily extract information from the corresponding

parts of the ciphertexts within the special time slot. From the above discussions, we conclude that simply extending existing schemes may not provide a post-compromise-secure solution.

B. Discussions on Type 2 Threat Model

Following the philosophy of DSSE, we say that a server should be honest-but-curious (i.e., under Type 1 model). But it becomes extremely powerful in Type 2 model. This yields further impossibilities in the design. A natural concern is how we could detect and resist data injection attacks launched by the server. Given the compromised secret key, the server can easily inject and tamper with the data in the database. All changes made by the server are now essentially "valid" due to the knowledge of the key. It could be feasible for the client to locally maintain extra verification information of the encrypted database so as to detect any illegal operations on the database, like [11]. This approach may significantly increase client-side storage, computation, and communication costs. Further, it is required that the verification information should be stored separately from the secret key on the client side. And so far there is no evidence that this verifiable approach is practically secure. For example, the server is able to obtain the state information (including the verification knowledge, e.g., how the verification is done) of the encrypted database with the compromised key. With that knowledge, the server may adaptively perform malicious operations which can bypass the verification.

Another challenge is to guarantee the security of the keyword search after a key compromise. Once the key compromise happens, the server learns the exact search results and frequencies of all the keywords stored in the encrypted database. The leaked information can be exploited to infer the underlying keyword of the subsequent client's keyword queries [14], [58], [59], even after the KeyUpdate. Recall that there is a special time slot, the period after the key compromise and before the key update. In this slot, we still need to protect the ciphertexts and keyword search queries. The techniques used to design volume-hiding structured encryption [42] and query-equality-suppressing structured encryption [34], [43] may be the potential solutions. But it is unknown if it is possible to apply them to the oblivious KeyUpdate. How to design a secure and practical scheme in Type 2 threat model is an interesting problem.

C. Discussions on Bamboo Construction

File Identifier Length. The construction of Bamboo relies on the DDH assumption, and the file identifier is encoded into a cyclic group element. This may restrict the file identifier length. Fortunately, this problem is quite easy to solve. Specifically, to support a long file identifier, one can split the identifier into small pieces so that each piece can fit the length limitation of a group element. Those pieces are encoded into elements and then can be respectively encrypted with the same random number and secret key and different hash functions. After running Bamboo.DataUpdate, those encrypted elements are viewed as a whole file identifier ciphertext and uploaded to the server. Later, the client can decrypt all the elements and merge the pieces to recover the identifier. It is easy to see that the above approach does not affect security.

File Deletion. When handling a deletion request on a pair of keyword and file identifier, the DataUpdate may encrypt the operation type $op = del$ with the pair to generate a ciphertext as a special deletion query. This approach logically marks the pair as "deleted" but does not remove it from $\mathbf{EDB}$. This may not be a "completed" deletion for the pair. A similar method is used in some existing DSSE, e.g., MITRA. The reason behind the design is that DataUpdate should not leak anything in the setting of post-compromise security. We also may not employ the Search to remove the deletion (unlike Aura) since any two Search queries must be indistinguishable in the view of the thief. A possible enhancement could be to enable KeyUpdate to locate and remove the ciphertexts and further update the key of the remaining ciphertexts. However, it may be challenging to reduce the information leakage to capture the post-compromise security in this context fully. We leave this challenge for future research.

KeyUpdate Intervals. Theoretically, the more we execute KeyUpdate, the better we achieve key-compromise security. In practice, we may use three strategies to balance KeyUpdate and security: (1) follow the suggestions given by standards, e.g., NIST Special Publication 800-57 [4] - the symmetric data-encryption key should be updated within 1-2 years after being created; (2) when the client detects/suspects the secret key is (partially) leaked; (3) when the encrypted database stays idle for a certain period. In fact, the KeyUpdate overhead is practical. For example, updating the key over the database with about 3,200,000 entries (using two threads) costs roughly 50 minutes. This overhead is mostly on the server side, while the client needs less than 30 microseconds.

Extension to Type 2 Threat Model. Bamboo is provably secure under Type 1 threat model. Unfortunately, it is so far impossible to extend it in Type 2 threat model efficiently. As explained in Section VII-B, the security under Type 2 threat model relies on the robust verification mechanism and oblivious KeyUpdate. The difficulty is in the design of the latter. The current design of KeyUpdate cannot provide obliviousness since the server uses the KeyUpdate token to update the encrypted database ciphertext by ciphertext. A straightforward solution is to require the client to download, decrypt, re-encrypt, and re-upload the whole database. This may be extremely costly and does not scale well in practice. A practical oblivious KeyUpdate without strong security assumptions (e.g., relying on a trusted third party) and expensive costs remains an open question.

Extension to Multi-Keyword Search. SEKU is formalized based on the classic DSSE definition under the single-keyword search context. Fortunately, we can extend Bamboo to support multi-keyword conjunctive search by a *cross-tag* technique [16], [49], [60]. We use two types of encrypted databases on the server side. One is the traditional encrypted database as in DSSE (named **TSet**), and the other is to verify the conjunctive relationships between two given keywords (named **XSet**). In a DataUpdate with an entry $(op, ((w_1, w_2, ..., w_k), id))$, we add/delete the records $(w_1, id), (w_2, id), ..., (w_k, id)$ to/from **TSet**, and further construct special tags of the entry to store in **XSet**. Whilst handling a conjunctive query $w_1 \wedge w_2 \wedge ... \wedge w_n$, we query **TSet** to get the file identifiers to w_1 and then use **XSet** to verify if each of the returned identifiers contains $w_2, w_3, .., w_n$. One may follow the above approach to instantiate **TSet** with

Bamboo and design a post-compromise-secure **XSet**.

Against Inference Attacks. In the context of key compromise, provided that the encrypted database can be exposed to the thief, one may ask if Bamboo is vulnerable to the inference attacks [6], [14], [39], [59]. This type of attack should leverage a sufficient amount of the leakage from the Search. From the thief's perspective, we say that it cannot see the Search leakage as the client and server communicate via a secure channel by DH exchange; meanwhile, it does not collude with the server. But if the server actively launches the attack, Bamboo may not perform well, as prior forward and Type-II backward secure DSSE schemes (e.g., Fides [12], MITRA [18], Aura [65]). We note that they leak the same amount of information to the server during Search. Existing practical DSSE schemes with backward security also cannot counter the inference attack, as they (by definition) leak the search results and (partial) access pattern to the server. To mitigate the attack, we may straightforwardly apply a current countermeasure to Bamboo, e.g., volume hiding solution [42] and leakage suppression strategy [34], which may produce extra overhead during both DataUpdate and Search on the client side.

Multiple Clients. Some research works [37], [64], [67] enable multiple clients to collaboratively write/read an encrypted database with fine-grained access control. It is non-trivial for Bamboo to support this. The main challenge is in modeling security. Given multiple data owners/users, the role of the thief and all the cases of "key stealing" should be carefully defined. For example, the thief could be among the clients with only write permissions, or it compromises those with both write and read rights; and it may further collude other clients to collect sensitive information from the database. We leave the multi-client case as an open problem.

D. Discussions on Experiments

Other Databases. In the experiment, we used the Postgresql database to store the generated ciphertexts for the compared schemes. One may choose to use other databases, e.g., MySQL or MongoDB. We state that the experimental results while using other databases could be slightly different, because they may lead to different overheads when accessing the ciphertexts. We leave this to the interested readers.

Keywords. One may argue that the 25 keywords used in the test dataset may not produce comprehensive experiments. In fact, the total size of the test dataset is 3,257,613, and the keywords we used can sufficiently show the performance differences among the compared schemes. According to the performance variation tendency, we state that using more keywords will not change the conclusion of our current experiments.

Network. We argue that the real-world network environment is usually influenced by many uncontrollable and unforeseen factors, such as burst network traffics or relay router failure. Those factors may make the network hard to set up a stable reproducible experimental foundation providing a fair comparison among the schemes. The experimental network environment was simulated over a stable LAN network, the network delay was artificially produced via the "tc" command, and all tested elements can be under our control. Thus, the simulated network is more beneficial for us to create a relatively fair test environment.

VIII. DSSE Revisited and Related Works

Kamara et al. [45] formally defined the syntax and adaptive security for DSSE. The security concentrates on the information leakage [24] revealed to the server. Since then, many DSSE schemes have been proposed to achieve high search efficiency [36], [44], supporting scalable database [15], physical deletion [70], and retaining small leakage [63].

Zhang et al. [72] proposed the well-known file-injection attack against DSSE. This attack enables the adversary to actively inject crafted files into the encrypted database to infer the underlying keywords of search queries. As an effective countermeasure to this attack, forward security has been considered as an essential property, which requires that a newly updated ciphertext leaks nothing about its underlying keyword. The first forward secure DSSE scheme was proposed by Chang et al. [19]. Later, Stefanov et al. [63] formalized the forward security using the leakage functions. After that, many forward secure DSSE schemes have been constructed to deliver sub-linear search complexity [10], small leakage [32], high practical performance [46], and high I/O efficiency [62].

Another important feature of DSSE, called backward security, was defined by Stefanov et al. [63]. It restricts the information leakage about deleted ciphertexts during search queries. Bost et al. [12] formalized three types of backward security with leakage functions. Since then, many research works have proposed forward and backward secure constructions to achieve small search leakage [18], [52], [73], robustness under fault operations [71], constant client storage [25], [38], and practical search performance [17], [21], [65], [66].

There are other works on searchable symmetric encryption, e.g., using trusted hardware to reduce the leakage from the server [1], [56], improving the I/O performance of the encrypted database [9], [26], [55], and enabling conjunctive search [49], [60], [69], [74].

All the aforementioned works have an implicit but unrealistic security assumption that the client's secret key will not be compromised. Once this assumption does not hold, all prior schemes become insecure. No prior works systematically investigates the key compromise problem and the countermeasures. This paper contributes to this line of research by developing post-compromise security for DSSE.

IX. Conclusions

We investigated and initialized the research topic of DSSE with KeyUpdate. We defined the notion SEKU and formulated the post-compromise security against Type 1 model. We further constructed the first scheme of its type, the post-compromise-secure instantiation Bamboo, and meanwhile proved its security. We state that the post-compromise feature may be a practical consideration for real-world applications. For example, the client may temporarily use a third-party device (e.g., a public computer) to query the encrypted database. This may risk the exposure of the secret key. Bamboo may provide an accountable solution that existing DSSE schemes cannot. As for efficiency, Bamboo can achieve sub-linear

search complexity and constant client time cost. Finally, we evaluated Bamboo with a real-world dataset. The experimental results show that Bamboo achieves a comparable search performance to the well-studied forward-and-backward secure DSSE schemes and offers high performance in KeyUpdate and client complexity. To further improve bandwidth, we introduced a flexible padding technique and then leveraged it to construct Bamboo*, which significantly outperforms Bamboo, especially in a large-scale database where there are many keywords with a small size of search results.

Acknowledgements. We would like to thank the shepherd Prof. Gang Qu and the anonymous reviewers for their valuable comments. This work was partly supported by the National Key Research and Development Program of China under Grant No. 2021YFB3101304, the Wuhan Applied Foundational Frontier Project under Grant No. 2020010601012188, the National Natural Science Foundation of China under Grant No. 62272186 and No. 61872412, and the Guangdong Provincial Key Research and Development Plan Project under Grant No. 2019B010139001. This work was also supported by EU Horizon research and innovation programme under grant agreement No. 952697 (ASSURED), No. 101021727 (IRIS) and No. 101070052 (TANGO).

REFERENCES

[1] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with SGX," in *EuroSys 2019*. ACM, 2019, pp. 4:1–4:6.

[2] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao, "RELIC is an Efficient LIbrary for Cryptography," https://github.com/relic-toolkit/relic, 2020, accessed April 04, 2022.

[3] G. Attardi, "Wikiextractor," https://github.com/attardi/wikiextractor, 2021, accessed April 04, 2022.

[4] E. Barker, "Recommendation for key management: Part 1 – general," 2020, accessed: May 22, 2021.

[5] Bitglass, "Next-gen CASB searchable encryption," 2022, accessed: April 04, 2022. [Online]. Available: https://www.bitglass.com/cloud-encryption

[6] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks," in *NDSS 2020*. The Internet Society, 2020.

[7] D. Boneh, "The decision diffie-hellman problem," in *ANTS 1998*, J. Buhler, Ed., vol. 1423, 1998, pp. 48–63.

[8] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan, "Key homomorphic prfs and their applications," in *CRYPTO 2013*, vol. 8042, 2013, pp. 410–428.

[9] A. Bossuat, R. Bost, P. Fouque, B. Minaud, and M. Reichle, "SSE and SSD: page-efficient searchable symmetric encryption," in *CRYPTO 2021*, vol. 12827, pp. 157–184.

[10] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *CCS 2016*, 2016, pp. 1143–1154.

[11] R. Bost, P. Fouque, and D. Pointcheval, "Verifiable dynamic symmetric searchable encryption: Optimality and forward security," *IACR Cryptol. ePrint Arch.*, p. 62, 2016. [Online]. Available: http://eprint.iacr.org/2016/062

[12] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *CCS 2017*, 2017, pp. 1465–1482.

[13] C. Boyd, G. T. Davies, K. Gjøsteen, and Y. Jiang, "Fast and secure updatable encryption," in *CRYPTO 2020*, vol. 12170, 2020, pp. 464–493.

[14] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *ACM SIGSAC 2015*, I. Ray, N. Li, and C. Kruegel, Eds., 2015, pp. 668–679.

[15] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS 2014*, 2014.

[16] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO 2013*, vol. 8042, 2013, pp. 353–373.

[17] J. G. Chamani, D. Papadopoulos, M. Karbasforushan, and I. Demertzis, "Dynamic searchable encryption with optimal search in the presence of deletions," in *USENIX Security 2022*, 2022, pp. 1–1. [Online]. Available: https://eprint.iacr.org/2022/648

[18] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *CCS 2018*, 2018, pp. 1038–1055.

[19] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *ACNS 2005*, vol. 3531, 2005, pp. 442–455.

[20] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *ASIACRYPT 2010*, vol. 6477, 2010, pp. 577–594.

[21] T. Chen, P. Xu, W. Wang, Y. Zheng, W. Susilo, and H. Jin, "Bestie: Very practical searchable encryption with forward and backward security," in *ESORICS 2021*, vol. 12973, 2021, pp. 3–23.

[22] P. S. S. Concil, "Requirements and security assessment procedures v3.2.1," 2018, accessed: May 22, 2021. [Online]. Available: https://www.pcisecuritystandards.org/document_library

[23] T. S. Consortium, "Sqlite home page," https://www.sqlite.org/index.html, 2022, accessed April 04, 2022.

[24] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *ACM 2006*, 2006, pp. 79–88.

[25] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," 2020.

[26] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," in *CRYPTO 2018*, vol. 10991. Springer, 2018, pp. 371–406.

[27] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "SEAL: attack mitigation for encrypted databases via adjustable leakage," in *USENIX Security 2020*. USENIX Association, 2020, pp. 2433–2450.

[28] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[29] F. S. Foundation, "The gnu mp bignum library," https://gmplib.org/, 2022, accessed April 04, 2022.

[30] O. S. Foundation, "Openssl," https://www.openssl.org/, 2022, accessed April 04, 2022.

[31] W. Foundation, "Wikimedia downloads," 2022, accessed April 04, 2022. [Online]. Available: https://dumps.wikimedia.org/enwiki/20220401/

[32] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption," in *CRYPTO 2016*, vol. 9816, 2016, pp. 563–592.

[33] R. Gay, A. Jain, H. Lin, and A. Sahai, "Indistinguishability obfuscation from simple-to-state hard problems: New assumptions, new techniques, and simplification," in *EUROCRYPT 2021*, A. Canteaut and F. Standaert, Eds., vol. 12698. Springer, 2021, pp. 97–126.

[34] M. George, S. Kamara, and T. Moataz, "Structured encryption and dynamic leakage suppression," in *EUROCRYPT 2021*, A. Canteaut and F. Standaert, Eds., vol. 12698, 2021, pp. 370–396.

[35] T. P. G. D. Group, "Postgresql: The world's most advanced open source database," https://www.postgresql.org/, 2022, accessed April 04, 2022.

[36] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *ACM CCS 2014*. ACM, 2014, pp. 310–320.

[37] A. Hamlin, A. Shelat, M. Weiss, and D. Wichs, "Multi-key searchable encryption, revisited," in *PKC 2018*, vol. 10769, 2018, pp. 95–124.

[38] K. He, J. Chen, Q. Zhou, R. Du, and Y. Xiang, "Secure dynamic searchable symmetric encryption with constant client storage cost," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 1538–1549, 2021.

[39] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *NDSS 2012*, 2012.

[40] A. Jain, H. Lin, and A. Sahai, "Indistinguishability obfuscation from well-founded assumptions," in *STOC 2021*, S. Khuller and V. V. Williams, Eds. ACM, 2021, pp. 60–73.

[41] Y. Jiang, "The direction of updatable encryption does not matter much," in *ASIACRYPT 2020*, vol. 12493, 2020, pp. 529–558.

[42] S. Kamara and T. Moataz, "Computationally volume-hiding structured encryption," in *EUROCRYPT 2019*, vol. 11477, 2019, pp. 183–213.

[43] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured encryption and leakage suppression," in *CRYPTO 2018*, H. Shacham and A. Boldyreva, Eds., vol. 10991, 2018, pp. 339–370.

[44] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *FC 2013*, vol. 7859, 2013, pp. 258–274.

[45] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *CCS 2012*, 2012, pp. 965–976.

[46] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *CCS 2017*, pp. 1449–1463.

[47] M. Klooß, A. Lehmann, and A. Rupp, "(R)CCA secure updatable encryption with integrity protection," in *EUROCRYPT 2019*, vol. 11476, 2019, pp. 68–99.

[48] C. Labs, "Acra database security," 2022, accessed: April 04, 2022. [Online]. Available: https://www.cossacklabs.com/acra/

[49] S. Lai, S. Patranabis, A. Sakzad, J. K. Liu, D. Mukhopadhyay, R. Steinfeld, S. Sun, D. Liu, and C. Zuo, "Result pattern hiding searchable encryption for conjunctive queries," in *CCS 2018*, 2018, pp. 745–762.

[50] A. Lehmann and B. Tackmann, "Updatable encryption with post-compromise security," in *EUROCRYPT 2018*, vol. 10822, 2018, pp. 685–716.

[51] J. Leyden, "23,000 https certs will be axed in next 24 hours after private keys leak," 2018, accessed: April 04, 2022. [Online]. Available: https://www.theregister.com/2018/03/01/trustico_digicert_symantec_spat/

[52] J. Li, Y. Huang, Y. Wei, S. Lv, Z. Liu, C. Dong, and W. Lou, "Searchable symmetric encryption with forward search privacy," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 1, pp. 460–474, 2021.

[53] Lookout, "Lookout casb," 2021, accessed: April 04, 2022. [Online]. Available: https://www.lookout.com/documents/whitepapers/us/lookout-casb-platform-overview-wp-us.pdf

[54] McAfee, "MVISION cloud for salesforce security," 2022, accessed: April 04, 2022. [Online]. Available: https://www.mcafee.com/enterprise/en-us/products/mvision-cloud/salesforce.html

[55] I. Miers and P. Mohassel, "IO-DSSE: scaling dynamic searchable encryption to millions of indexes by improving locality," in *NDSS 2017*. The Internet Society, 2017.

[56] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE S&P 2018*. IEEE Computer Society, 2018, pp. 279–296.

[57] P. Muncaster, "Stolen cloud api key to blame for imperva breach," 2019, accessed: April 04, 2022. [Online]. Available: https://www.infosecurity-magazine.com/news/stolen-cloud-api-key-to-blame-for/

[58] J. Ning, X. Huang, G. S. Poh, J. Yuan, Y. Li, J. Weng, and R. H. Deng, "LEAP: leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset," in *CCS 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds., 2021, pp. 2307–2320.

[59] S. Oya and F. Kerschbaum, "Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption," in *USENIX Security 2021*, M. Bailey and R. Greenstadt, Eds., 2021, pp. 127–142.

[60] S. Patranabis and D. Mukhopadhyay, "Forward and backward private conjunctive searchable symmetric encryption," in *NDSS 2021*, 2021.

[61] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[62] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized I/O efficiency," *IEEE Trans. Dependable Secur. Comput.*, vol. 17, no. 5, pp. 912–927, 2020.

[63] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *NDSS 2014*, 2014.

[64] S. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for boolean queries," in *ESORICS 2016*, vol. 9878, pp. 154–172.

[65] S. Sun, R. Steinfeld, S. Lai, X. Yuan, A. Sakzad, J. K. Liu, S. Nepal, and D. Gu, "Practical non-interactive searchable encryption with forward and backward privacy," in *NDSS 2021*, 2021.

[66] S. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *CCS 2018*, 2018, pp. 763–780.

[67] J. Wang and S. S. M. Chow, "Omnes pro uno: Practical multi-writer encrypted database," in *USENIX 2022*, 2022, pp. 2371–2388.

[68] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *CCS 2014*, 2014, pp. 215–226.

[69] Z. Wu and K. Li, "Vbtree: forward secure conjunctive queries over encrypted data for cloud computing," *VLDB J.*, vol. 28, no. 1, pp. 25–46, 2019.

[70] P. Xu, S. Liang, W. Wang, W. Susilo, Q. Wu, and H. Jin, "Dynamic searchable symmetric encryption with physical deletion and small leakage," in *ACISP 2017*, vol. 10342, 2017, pp. 207–226.

[71] P. Xu, W. Susilo, W. Wang, T. Chen, Q. Wu, K. Liang, and H. Jin, "Rose: Robust searchable encryption with forward and backward security," *IEEE TIFS*, vol. 17, pp. 1115–1130, 2022.

[72] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security 2016*, 2016, pp. 707–720.

[73] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *ESORICS 2019*, vol. 11736, 2019, pp. 283–303.

[74] C. Zuo, S. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and G. Wei, "Forward and backward private dynamic searchable symmetric encryption for conjunctive queries," *IACR Cryptol. ePrint Arch.*, p. 1357, 2020.

APPENDIX A
DDH ASSUMPTION

Definition 5 (Decisional Diffie-Hellman (DDH) Assumption)**.** *Let $\mathbb{G}$ be a multiplicative cyclic group of prime order q, and g is a generator of $\mathbb{G}$ where q is of λ bit-length. We say that the DDH assumption holds in $\mathbb{G}$ if for any PPT adversary $\mathcal{A}$ the probability that $\mathcal{A}$ distinguishes between tuples (g, g^a, g^b, g^{ab}) and (g, g^a, g^b, g^c) is negligible in λ where $(a, b, c) \xleftarrow{\$} \mathbb{Z}_q^* \times \mathbb{Z}_q^* \times \mathbb{Z}_q^*$.*