REARGUARD: Secure Keyword Search Using Trusted Hardware

Wenhai Sun, Ruide Zhang, Wenjing Lou, and Y. Thomas Hou
Virginia Polytechnic Institute and State University, Blacksburg, VA, USA

***Abstract*—Search over encrypted data (SE) enables a client to delegate his search task to a third-party server that hosts a collection of encrypted documents while still guaranteeing some measure of query privacy. Software-based solutions using diverse cryptographic primitives have been extensively explored, leading to a rich set of secure search indexes and algorithm designs. However, each scheme can only implement a small subset of information retrieval (IR) functions and often with considerable search information leaked. Recently, the hardware-based secure execution has emerged as an effective mechanism to securely execute programs in an untrusted software environment. In this paper, we exploit the hardware-based execution environment (TEE) and explore a software and hardware combined approach to address the challenging secure search problem. For functionality, our design can support the same spectrum of plaintext IR functions. For security, we present oblivious keyword search techniques to mitigate the index search trace leakage. We build a prototype of the system using Intel SGX. We demonstrate that the proposed system provides broad support of a variety of search functions and achieves computation efficiency comparable to plaintext data search with elevated security protection.**

I. Introduction

Nearly two decades has passed since Song *et al.*'s seminal work on the first encrypted data search scheme [1]. This demonstrated that the fascinating concept of retrieving information from encrypted data can be accomplished using cryptography. Since then, SE has received a growing interest from both academia [2], [3], [4], [5], [6] and industry [7]. Recently, the importance of this technique has been highlighted due to the advent of cloud computing, where there is a strong desire to protect users' sensitive information from prying eyes while providing fundamental data services.

There are two main research directions in achieving the grand vision of search over encrypted data. One is software-based secure computation research, which often relies on cryptography and focuses on algorithmic design and theoretical proof. The other is the trusted execution solutions that depend on hardware isolation and trusted computing. The conventional SE is realized using software-based solutions. Albeit there are extensive investigations along this research line, current SE realization is not satisfactory in two aspects. First is the obvious query function gap between SE and the plaintext IR technology. This is because efficient practical SE solutions are built on top of a variety of crypto primitives, such as property-preserving encryption [8], functional encryption [6], and searchable symmetric encryption (SSE) [2], and each crypto tool only supports a specific class of query types by incorporating different index structures and search algorithms. In addition, existing realistic SE solutions have many security limitations. In the symmetric setting, the most secure SSE we can achieve is under the L_1 leakage profile [9], which at least reveals the index search trace, including search pattern and access pattern (see Sect. II). Unfortunately, the once considered "inconsequential" information disclosure has not been well studied and already led to many devastating attacks in practice [9], [10]. Besides the above information leakage, public-key based schemes are inherently vulnerable to predicate privacy breach [11], i.e. an adversary can generate ciphertexts with the public key and infer the queried keywords during the search process.

On the other hand, hardware-based trusted execution environment has recently emerged as an effective security mechanism in achieving trustworthy execution of applications [12], [13], [14]. These systems adopt trusted hardware, such as Trusted Platform Module (TPM), Intel Trusted Execution Technology (TXT), ARM TrustZone, Intel Software Guard Extensions (SGX) and a small size of firmware as the trusted computing base (TCB). This TCB provides not only the root of trust but also the necessary system isolation for the environment. While it might appear that one can simply migrate the state-of-the-art IR techniques into the TEE to enable the same spectrum of query functions with enhanced security, there are several challenges that require careful design considerations to take advantage of the technology.

While the hardware-based secure execution, such as Intel SGX, can provide confidentiality and integrity of the application inside the TEE, information side channel is often not protected [15], [16]. The threat is greatly amplified when users share resources with adversaries, yet resource sharing is the basis of cloud computing. Recently, both control channel [15] and cache channel [16] have been demonstrated to leak execution information on the Intel SGX platform. Thus, direct adoption of TEE for secure search applications [17], [18] can lead to the disclosure of the index search trace. Another challenge lies in the programming environment. In order to defend against the untrusted operating system, each library function potentially needs to be redesigned to harden the defense against attacks, such as Iago attack [19]. At the time of writing, there are a limited number of library functions available in the enclave, the TEE of Intel SGX. None of the IR software we studied can be directly adopted as an enclave library due to missing libraries from the version 1.6 of the Intel SGX SDK.

Our contributions. In this work, we tackle the fundamental yet challenging problem of search over encrypted data. We propose secu**r**e k**e**yword se**ar**ch usin**g** tr**u**sted h**ard**ware – REARGUARD, built on TEEs such as Intel SGX [20] and AMD Memory Encryption [21] to perform search computation completely within the isolated memory even if the privileged software is untrusted. Such hardware-enforced isolation provides the confidentiality and integrity of both data and computation, which is essential in cloud computing. Furthermore, REARGUARD enables IR functions comparable to plaintext data search. Our scheme is also a departure from the pure software-based approach whose computation overhead largely depends on the underlying cryptographic primitives.

Current practical designs of software-based SE have the leakage profile that reveals at least the index search trace. REARGUARD can achieve better information protection and significantly improve the security of SE by mitigating such leakage. We define and realize two new leakage profiles, L_0^+ and L_0, in the dynamic SE scenario supporting index update. In the L_0^+ model, we completely hide the index search trace and only reveal minimal information at the setup as prior work. In the weaker notion L_0, we allow some reasonable leakage for a more efficient query. We identify several query-dependent operations in the search algorithms and adapt them to keyword-oblivious executions to satisfy the defined security requirements. We prototype REARGUARD with 4,000 lines of code (LOC). Considerable efforts are made to ensure that the implementation meets the security requirement while at the same time offering the desired query functions. REARGUARD provides search functionality and performance comparable to the plaintext data search.

In summary, we make the following contributions in this study. 1) We propose REARGUARD, an innovative approach towards a dynamic secure keyword search scheme that employs the latest advancement in hardware-based secure execution – Intel SGX. The proposed system supports a rich set of query functions comparable to plaintext IR while ensuring the confidentiality and integrity of the query process. Our design is secure against strong attacks including those from compromised privileged software and low-level firmware. 2) Our design mitigates the index search trace leakage from the memory side channel by using oblivious keyword search functions. This is one of the major concerns about SE in the cloud where users and adversaries share resources. The system defines and realizes two leakage profiles to balance security and performance, both exhibiting substantially reduced index search footprints. 3) We carefully design and implement the popular IR functions into a fully-functional SGX-compatible prototype. Our experiment with the real-world dataset reports a performance close to that of plaintext data search with elevated security protection.

II. Background

This section provides background information on current SE leakage, inverted index structure, and Intel SGX.

Privacy leakage in SE. Among many cryptographic primitives for SE constructions, searchable symmetric encryption [2], [3], [4] is the most publicized and investigated in the literature. SSE exploits deterministic encryption for efficient keyword match but its protection of keyword privacy is weak. Cash *et al.* defined four leakage profiles, i.e. L_4, L_3, L_2 and L_1, for SSE in the static setting to characterize the amount of information leakage [9]. Specifically, L_4 schemes reveal the number of words, their orders and occurrence counts in a document. Examples include some commercial products [7]. L_3 profile reveals all the above information except occurrence counts of each keyword. L_2-SSE schemes only disclose the keyword number in a document. L_1 reveals the same information as L_2 but only for keywords that have been searched. Additionally, all the leakage profiles also imply the revelation of index search trace, including keyword *search pattern*, i.e. whether a query is repeated, and *access pattern*, i.e. pointers to encrypted files that satisfy the query. The majority of SSE adhere to L_2 or L_1 leakage. Such information disclosure also leads to *forward privacy* breach in the dynamic setting. This allows the adversary to learn whether the newly added document contains the keyword that has been queried before.

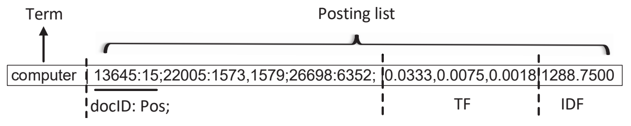


Fig. 1. Example of an inverted index row.

The consequence of the above leakage has not been well studied. Many attacks against SSE have emerged to exploit the leakage to partially or even fully recover the query and dataset information [9], [10]. Further, public-key based constructions are inherently vulnerable to predicate privacy leakage [11]. Namely, an adversary can generate ciphertexts with the public key and infer the query during the search process.

Inverted index. Inverted index is widely adopted in modern search engines and current SSE design. It enables sublinear search complexity (w.r.t. the number of files in the dataset), as well as rich query functions[1] in the dynamic setting, such as Boolean query, phrase query, ranked retrieval, spelling correction. It is generated by applying the standard indexing techniques to the target dataset [22], e.g. tokenization, stemming, stop words elimination, linguistic analysis, etc. In particular, this data structure contains N index rows vertically, where N equals the number of the extracted keywords (or terms, we use them interchangeably hereafter) from the dataset. Horizontally, it is divided into two major parts: a) the vocabulary (or dictionary) that includes all the extracted keywords, and b) the keyword-associated posting lists. Each list as shown in Fig. 1 is composed of all the identifiers docID[2] of the documents containing the keyword, term position Pos in the corresponding documents, and other statistics, e.g. term frequency (TF) and inverse document frequency (IDF), etc. In general, a query is performed over the index by matching the target keywords (Step 1), retrieving the associated posting

[1]Most SSE works only support simplified versions of inverted index for extremely constrained query types, such as single keyword search.

[2]They can be pointers/URLs to the documents.

lists (Step 2) and evaluating query functions via information from the acquired lists (Step 3). Note that in Step 1, we can either linearly scan the vocabulary or use the hash table with constant search cost. The observed leakage (see Sect. IV) exists in both cases. For simplicity, here we choose to traverse the vocabulary for keyword match.

Intel SGX. SGX is the latest Intel's instruction extensions that aim to offer integrity and confidentiality guarantees to security-sensitive computation conducted on the commodity computer. The privileged software and low-level firmware, such as OS kernel, virtual machine hypervisor, and system management mode, are all assumed to be potentially malicious in the adversary model of SGX [20]. This is because the TCB of SGX, compared to its predecessors, e.g., TPM, Intel TXT, only contains the CPU and several privileged *enclaves*. This significantly reduces the attack surface and provides strong security guarantees. The memory region reserved for the enclave, called enclave page cache (EPC), can only be accessed when CPU enters the special *enclave mode*, which enforces additional hardware checks on every external EPC page access request. The EPC memory is encrypted and authenticated by SGX Memory Encryption Engine (MEE) [23], part of the memory controller within the CPU package. Enclave can save the encrypted computation result onto the untrusted persistent storage by using symmetric authenticated encryption, such as GCM[AES]. The encryption key is derived from the hard-coded *root sealing key* unknown to Intel. The ciphertext can be loaded back and decrypted later by the same enclave that encrypts it. In addition, SGX also provides the remote attestation function to convince users of the integrity of the established enclave before setting up the secure channel and provisioning their secrets.

Besides physical attacks, SGX is also vulnerable to rollback attack [23], Iago attack [19] and side-channel attacks, including cache timing, power analysis, etc. SGX cannot defend against DoS attack as the underlying resource allocation is still controlled by the privileged system software. When applying these attacks to an SGX-based search, the adversary can interrupt the search process or/and infer the search privacy by breaking the SGX protection. This is out of scope of this paper. Further, memory trace leakage has been confirmed at both page [15] and cache line level [16]. The leakage will disclose index search trace to the adversary in SE. We aim to mitigate such information disclosure in this work.

III. Problem Formulation

A. Overview

Three entities, *data owner*, *data user* and *SGX-enabled server* are involved in the system as shown in Fig. 2. Data owner possesses a collection of n documents (we use "document" or "file" to refer to any text content records, such as text files, web pages), $\mathbf{DB} = \{D_1, D_2, ..., D_n\}$, which in turn contain m keywords $\mathbf{W} = \{w_1, w_2, ..., w_m\}$. Then he generates an inverted index $\mathcal{I}$ for $\mathbf{DB}$. Consistent with SSE [2], [5] we decouple the storage of the dataset from the storage

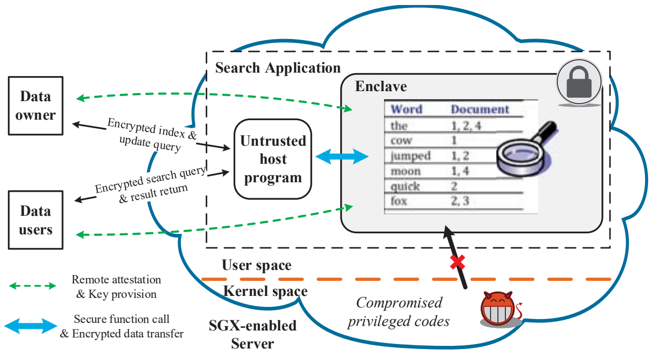

Search Application
Enclave
Word
Document
the
1, 2, 4
cow
1
jumped
1, 2
moon
1, 4
quick
2
fox
2, 3
Data owner
Encrypted index & update query
Untrusted host program
Encrypted search query & result return
Data users
User space
Kernel space
Remote attestation & Key provision
Secure function call & Encrypted data transfer
SGX-enabled Server
Compromised privileged codes

Fig. 2. REARGUARD framework

of its index[3]. We assume that the code of query algorithms and associated parameters are public and have been preloaded into the enclave that is set up by the server. Next, the data owner authenticates himself to the server and launches remote attestation to check the integrity of the code and static data in the enclave. Then he establishes a secure channel and sends an owner-generated secret key sk_o to the enclave. The data owner also generates index ciphertext $\tilde{\mathcal{I}}$ under sk_o, for instance using GCM[AES], and pass it to the server. Later he, under the same sk_o[4], can issue an encrypted index update request τ_{upd} to add or delete a document.

By going through a similar procedure, an authorized data user i verifies the enclave that hosts the search code by remote attestation. This guarantees the integrity of the execution of the search program and correctness of the query result. The user also shares his secret key sk_{u_i} with the enclave and uploads the query ciphertext τ_s under sk_{u_i} to the server.

On the server side, search process begins with enclave loading and decrypting index $\tilde{\mathcal{I}}$ and query τ_s with the corresponding keys. The query is executed over the plaintext index inside the enclave. The ciphertext $\tilde{res}$ of the result documents id using sk_{u_i} is sent back to user i. Another advantage of our design over SSE is that it naturally supports the more realistic multi-user setting because each user is able to search by his own secret key shared with the enclave.

Definition 1: (REARGUARD) Our secure keyword search scheme using trusted hardware is a tuple of three protocols executed between the data owner, data users and the SGX-enable server as follows:

- $(sk_o, \tilde{\mathcal{I}}) \leftarrow$ Setup$(1^\lambda, \mathcal{I})$: On input a security parameter λ and an inverted index $\mathcal{I}$ for a dataset, it for the data owner outputs a secret key sk_o that will be shared with the verified enclave on the server and the encrypted index structure $\tilde{\mathcal{I}}$ that will be stored on the server.
- $(\tilde{res}, \tau_s, sk_{u_i}) \leftarrow$ Search$(1^\lambda, Q, \tilde{\mathcal{I}}, sk_o)$: On input the security parameter λ, it outputs for the user i a secret key sk_{u_i} that will be shared with the attested enclave on the server. Using sk_{u_i}, it encrypts a query Q on some keywords w into ciphertext τ_s, which is sent to the server.

[3]This is a common practice. For example, Google is only responsible for index searching and maintaining, not hosting the actual web contents.

[4]A different secret key can be generated for each interaction with the server.

On input τ_s, $\tilde{\mathcal{I}}$, sk_o, sk_{u_i}, it outputs the search result ciphertext $\tilde{res}$ under sk_{u_i} for the user.

- $(\tilde{\mathcal{I}}_\Delta, \tau_{upd}) \leftarrow$ Update$(1^\lambda, upd, \tilde{\mathcal{I}}, sk_o)$: On input an index update request $upd = \{add/delete, w, id\}$ (perform addition or deletion of the file id over the posting list of keyword w) and the owner's secret key sk_o, it outputs an update token τ_{upd}. On input τ_{upd}, $\tilde{\mathcal{I}}$, sk_o, it produces an updated index $\tilde{\mathcal{I}}_\Delta$.

B. Adversary Model

We identify several keyword-dependent query operations and propose oblivious index access techniques to ensure that the adversary who observes a sequence of memory accesses toward the index, including the addresses and encrypted contents, has an indistinguishable view on index search (update) operations from the exhibited memory traces given two search (update) queries. Our security assumption is consistent with SGX except that we extend the side-channel attacks to comprise any attacks using information not derived directly from index access, such as target dataset statistics (e.g. posting list length, keyword frequency, etc.), context information (e.g. trending words, communication volume), and knowledge of linguistics. We do not intend to defend against them in this paper. We also aim to achieve *query unlinkability*, i.e. the adversary cannot distinguish search tokens only by their appearances even for the same keywords, which is not supported by most SSE. In what follows, we first define two leakage profiles, L_0^+ and L_0 and then give our security definition.

Definition 2: (L_0^+ – Complete index access trace hiding) It reveals the initial index size at the setup phase, deterministic index search trace and update pattern of the same operation (add or delete) for any keyword.

Definition 3: (L_0 – Partial index access trace hiding) This profile reveals the initial index size at the setup phase, deterministic index search trace and update pattern of the same operation for keywords in the same group (see Sect. IV).

Similar to SSE, we do not consider index access operation types, i.e. search or update, to be sensitive information, albeit they can be further protected at extra cost [24].

Security definition. We define the security of our scheme based on the simulation model of the secure computation [4]. In particular, it requires that a real-world protocol execution $\Pi_\mathcal{F}$ using the secure hardware functions be able to simulate an ideal-world functionality $\mathcal{F}$, such that an environment $\mathcal{Z}$, who produces all the input and reads all the output in the system, cannot distinguish these two worlds. We define the experiments $\mathbf{Real}_{\Pi_\mathcal{F},\mathcal{A},\mathcal{Z}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{F},\mathcal{A},\mathcal{S},\mathcal{Z}}(\lambda)$ in real world and ideal world respectively as follows based on both leakage profiles L_0^+ and L_0.

$\mathbf{Real}_{\Pi_\mathcal{F},\mathcal{A},\mathcal{Z}}(\lambda)$: In the setup phase, an environment $\mathcal{Z}$ instructs the data owner by sending him a "setup" message to perform the Setup protocol with the real-world adversary $\mathcal{A}$. In each time step, $\mathcal{Z}$ specifies a search query Q for the user or an update request $upd = \{add/delete, w, id\}$ for the data owner. The user (owner) executes Search (Update) protocol. $\mathcal{Z}$ observes the protocol output for each search (update) operation, which is either a protocol abortion $\perp$, search result, or update success. Finally, it outputs a bit b.

$\mathbf{Ideal}_{\mathcal{F},\mathcal{A},\mathcal{S},\mathcal{Z}}(\lambda)$: In the setup phase, an environment $\mathcal{Z}$ sends the data owner a message "setup". Then the owner forwards this message to an ideal functionality $\mathcal{F}$, which notifies an ideal-world adversary $\mathcal{S}$ of the leakage L_0^+ (L_0). In each time step, $\mathcal{Z}$ specifies a search query Q for the user or an update request $upd = \{add/delete, w, id\}$ for data owner. The user (owner) submits Q (upd) to $\mathcal{F}$. Then $\mathcal{S}$ is notified of the leakage L_0^+ (L_0) associated with the search (update) operation by $\mathcal{F}$. $\mathcal{S}$ sends $\mathcal{F}$ either "continue" or "abort". $\mathcal{F}$ outputs either search result, update success, or $\perp$, which is observed by the environment $\mathcal{Z}$. Finally, $\mathcal{Z}$ outputs a bit b'.

Definition 4: (Semi-honest/malicious security) We say that a protocol $\Pi_\mathcal{F}$ simulates the ideal functionality $\mathcal{F}$ in the semi-honest/malicious model, if for PPT semi-honest/malicious real-world adversary $\mathcal{A}$, there exists an ideal-world simulator $\mathcal{S}$, such that for all non-uniform, polynomial-time $\mathcal{Z}$,

$$|Pr[\mathbf{Real}_{\Pi_\mathcal{F},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - Pr[\mathbf{Ideal}_{\mathcal{F},\mathcal{A},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \leq \text{neg}(\lambda).$$

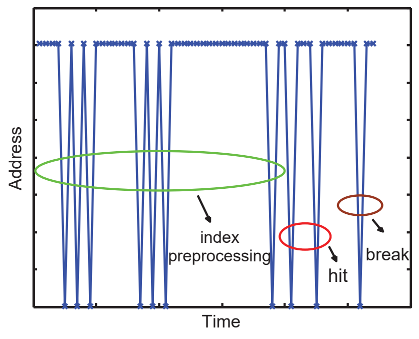
Our security definition covers both the semi-honest adversary who faithfully follows the prescribed protocol and the malicious adversary that arbitrarily deviates from the protocol. Privacy of the scheme is guaranteed because $\mathcal{S}$ is only given the leakage L_0^+ or L_0 during the simulation. The definition also captures the correctness as data user or owner in the ideal world receives either the expected result or a protocol abortion.

IV. Our Design

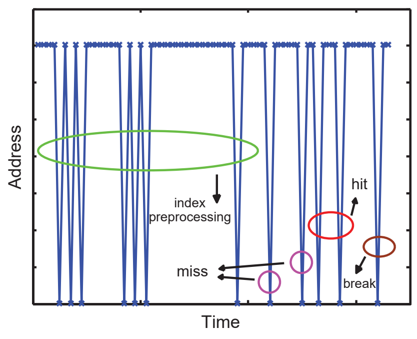
This section provides concrete design for REARGUARD, especially focusing on the Search and Update phases. We first deal with the fundamental single keyword query, which is extensively studied in SSE. Then we describe the dynamic setting and consider the scalability issue with SGX. In the end, we discuss extensions to other common query functions, such as spelling correction, Boolean query, phrase query, proximity query, range query and similarity-based rank retrieval.

A. Single Keyword Query

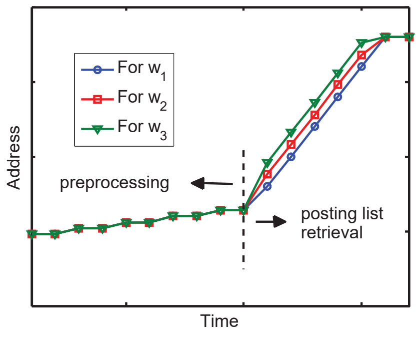
1) Index Search Trace Leakage: The successful execution of a single keyword query over an inverted index returns file IDs within the posting list of the intended keyword. We identify two keyword-dependent operations in the single keyword search algorithm in Fig. 4. The first sensitive operation is to search for the intended keyword w in the vocabulary of the index in Step 1 (see Sect. II). If and only if the condition is met (line 3), the corresponding code block in Step 2 will be executed to further retrieve the posting list of this keyword (line 4) and then terminate the index search (line 5). We also experimentally verify the existence of the leakage for different keywords using Intel Pin framework [25] in Fig. 3. Memory traces for different keyword matching in Step 1 are easily distinguished in Fig. 3 (a) (b). If the first keyword is intended as in Fig. 3 (a), the match hit can be observed followed by a break operation. In the case of the third keyword being the interest in Fig. 3 (b), we will first observe two misses, then the



(a)



(b)



(c)

Fig. 3. Memory trace for keyword match in Step 1: (a) The first keyword match; (b) The third keyword match. (c) Memory trace for posting list retrieval in Step 2 for the first three keywords.

```
Input: Query keyword $w$ and inverted index $\mathcal{I}$ including $m$ keywords, where $R_i$
       is the $i$th index row. Define the data structure $Rindex$ for $R_i$, where
       $Rindex.term$ is the term for this row and $Rindex.plist$ is the
       corresponding posting list with data structure $Plist$.
Output: Query result $res$.
1 $res = \emptyset$
2 for $i = 1 \to m$ do
3     if $R_i.term == w$ then        // Keyword-dependent condition
        evaluation
4         $res = R_i.plist$      // Keyword-dependent posting list
            retrieval
5         break
6     end
7 end
8 return $res$
```

Fig. 4. Pseudocode for single keyword query.

match hit and the break in the end[5]. In Step 2 of posting list retrieval, we can also differentiate the index search patterns for different keywords effortlessly in Fig. 3 (c). The leakage may still exist at different granularities even using SGX[15], [16]. As a result, hiding memory traces of these two query-dependent operations plays a critical role in our design.

2) Oblivious Keyword Search Primitives: In a nutshell, we implement oblivious keyword search primitives to obfuscate the memory traces during index access. The main idea is similar to the techniques used in [12], [26] but we tailor them for the purpose of secure keyword search. Specifically, we realize the oblivious data transfer by X86 CMOVZ instruction, which moves the source operand to the destination operand if the condition code is true. When both source and destination operands are put in registers, this data transfer turns out to be oblivious and leaks no information about the branch selection. Likewise, we are able to use CPU registers as private storage to conceal the search footprints. For an *oblivious read*, we first load contents into registers and then merely select the data of interest. On the other hand, an *oblivious write* operation is carried out as follows. It first obliviously read the content. Should the data be intended, the updated content will be stored; otherwise, the original data will be written back. Since SGX uses randomized encryption to protect every write operation by MEE, the adversary cannot infer the data content. Moreover, we observe that the index search process consists of a sequence of read operations and that we only need to write the index at the update phase. Note that it is unnecessary to further obfuscate search and update operations similar to SSE, while this can be done readily by additional dummy writes after oblivious reads.

[5]The miss may not be observable if we use a hash table, but we can still capture the leakage by the address of the hit.

Oblivious keyword match. We design an OMatch() function as shown in Fig. 5 to hide the trace from keyword match by using the aforementioned oblivious data transfer primitive. In particular, we store both the query and keyword in the vocabulary in separate registers (line 8) and then compare them (line 5). If there is a match, the pointer to the posting list of the queried keyword will be returned; otherwise, it will return a default dummy address (line 6).

Oblivious posting list retrieval. Another oblivious function ORetrieval() is also adopted in order to hide the index search trace for retrieving the posting list in Step 2. A posting list will be obliviously retrieved only when its address matches that returned by OMatch(); otherwise, ORetrieval() function will return a dummy list. In addition, we can further improve the efficiency of array reading by the vector register AVX2 instead of element-wise read using a general purpose register.

3) Put All Together: We will show the concrete design for leakage profiles L_0^+ and L_0 respectively using the proposed oblivious search functions.

L_0^+ construction. The main idea behind the construction for L_0^+ leakage profile is to display the deterministic index search trace for each query. Specifically, we first use OMatch() to scan the entire vocabulary and obliviously match the queried keyword. Then ORetrieval() function is executed to obtain the posting list of the intended keyword after touching every index row. We further obliviously pad the retrieved list for every query to the predefined length l, $l \geq \text{MaxLength}(plists)$, so as to further obscure the attacker's view. Despite the complexity $O(N)$, our hardware-based scheme is efficient in practice because of the fast protocol execution between the CPU registers and DRAM of the server.

L_0 construction. Our intention of creating L_0 profile is to speed up the search process by tolerating extra information leakage compared to L_0^+ but still to achieve better security than SSE. We first randomly divide the keyword universe $\mathbf{W}$ into groups and scan the index until the group including the intended keyword has been searched. Next, the posting

```
OMatch(Rindex* $rind$, Term $qterm$, Plist* $tmp$){
Plist* $match$;
  __asm__ volatile (
      "mov %3, %0;"
      "cmp %1, %2;"
      "cmovz %4, %0;"
      : "=r" ($match$)
      : "r" ($rind \to term$), "r" ($qterm$), "r" ($tmp$), "r" ($rind \to plist$)
      : "cc"
  );
return $match$;
}
```

Fig. 5. OMatch() wrapper.

list of interest is obliviously retrieved from the group. This construction results in a faster query process than L_0^+. Efficiency can be further improved by using additional constant-overhead data structures, such as hash table, Bloom filter, to allow direct search over the target group. We also pad the result list from group i to a preset length l_{g_i}, which is not shorter than the longest list in the group. This L_0 design reveals which part of the index is being queried. However, the adversary cannot differentiate two queries for the same group through the disclosed aggregated group search pattern. L_0^+ can be considered a special case of L_0 with only one group – the entire index. We provide the detailed discussion on the implication of group size in Sect. V.

4) Update: In the dynamic setting, the data owner is able to update the index by adding (deleting) a file to (from) the posting list of some keyword. We can leverage oblivious write operation to blur the view on the update. In particular, depending on the leakage profile, we first obliviously search over the index and obtain the intended posting list. For file addition, we insert the new file and its metadata to the retrieved list. Then we obliviously write the updated list back to the index, which increments the length of all the posting lists in the group for L_0 or in the entire index for L_0^+. Deleting a file for a keyword follows the similar procedure by replacing the target with a dummy file. In this case, the length of index rows is unchanged. The type of update operation, add or $delete$, is also revealed to the server by observing the size of the updated index. Albeit we do not consider the leakage sensitive, we can foil it by intentionally writing a dummy file to the corresponding lists for the deletion operation.

The proposed approach may cause gradually increased index storage over time. To address this issue, the data owner downloads the index after a predefined number of update operations. He then refreshes the index by deleting the dummy files and randomly shuffles index rows. He also regroups the index for L_0 before encrypting and uploading it to the server. As a result, the adversary only observes an aggregated update pattern. The view can be further obfuscated by randomly cleaning a portion of dummy files in the index.

5) Scalability: The problem with the straightforward implementation of REARGUARD is that the EPC memory is constrained by current SGX specification, i.e. 128MB in total. Our experiment shows only about 95MB available for code and data. This scalability problem affects all applications built on Intel SGX at present. In the wake of indexing a large dataset, the index size is likely to exceed the limitation. We circumvent this pressing issue by splitting the original large index into small partitions at the setup. These partition indexes when sitting outside enclave are protected by authenticated encryption, and loaded into enclave on demand. For L_0^+, all partitions are sequentially loaded and searched inside the enclave. For L_0, we adopt a hierarchical index structure for efficient on-demand loading. Specifically, we put a small first-level index (e.g. using hash table, Bloom filter) into the enclave and use it to quickly pinpoint the second-level index partition in the main memory containing the target group. Then the enclave loads and obliviously searches over the partition. We are also able to achieve faster search by dividing the original index as per the groups. As such, only the intended group index is fetched by the in-enclave primary index.

B. Additional Query Function Support

Besides the single keyword query, plaintext IR compasses a variety of functions. Due to the page limit, we briefly describe how to incorporate some popular functions into REARGUARD design.

Spelling correction has a wide implementation in modern search engines to provide users with correct search results even in the presence of misspellings in the query. REARGUARD can support this function by modifying OMatch(). Specifically, instead of exact keyword match, it checks whether the keyword being accessed is within the predefined distance, e.g. edit distance (ed), to the user input. If it is true, this keyword is obliviously selected as the correct query.

Boolean query is another fundamental query type used in database and free text search. The query is formed by concatenating multiple keywords with logical operations, such as AND, OR and NOT. Boolean query can be evaluated by performing set operations, i.e. intersection, union, and difference, in Step 3 based on the obliviously retrieved posting lists. Related data manipulation does not touch the index in this case. Thus, search trace will still be hidden.

Range query is extensively used in both database and free text search to match the records with terms within a certain range. We can either adopt a tree-based index in the enclave with equivalent security level of SSE [17], or transform the range query to a Boolean query [27] so as to achieve L_0/L_0^+ security. With the latter, we do not need to alter our base index structure and seamlessly support this query type.

We observe that many query functions are carried out in Step 3, the post index access phase. Proximity and phrase queries are common query types. In phrase query, all the matched documents should contain a particular sequence of keywords while proximity query constrains the result by specifying the allowed distances between queried keywords. These two functions can be treated as special cases of Boolean search with AND operation [22] and evaluated by using the Pos information in the retrieved posting lists. Similarity-based ranking is an advanced IR technique to rank result files by their relevance to the query using statistics of the dataset, for instance, the "TF $\times$ IDF" weight in the cosine measure of the vector space model [28]. We can calculate the similarity score after the posting lists are obliviously retrieved from the index.

V. Security Analysis

In this section, we prove the security of REARGUARD for single keyword search. The proofs for other query functions are similar due to the same protection methods.

Theorem 1: REARGUARD under L_0^+ is secure against the semi-honest adversary under Definition 4 if the underlying SGX primitives are trusted and encryption is CPA-secure.

Proof: (Sketch). In the setup phase, $\mathcal{S}$ can output an index $\mathcal{I}'$ with randomly generated index rows as per L_0^+. Then it simulates the encrypted index $\tilde{\mathcal{I}}' = Enc_{sk_o}(\mathcal{I}')$, where sk_o is randomly selected for the CPA-secure encryption Enc.

In the search phase, according to L_0^+, $\mathcal{S}$ randomly selects a keyword w' from $\mathcal{I}'$ as query Q'. τ_s' can be simulated by $Enc_{sk_u}(Q')$, where sk_u is randomly produced.

In the update phase, simulator $\mathcal{S}$ outputs $upd' = \{op, w', id'\}$ based on the leakage function L_0^+. Specifically, op is either add or $delete$ as per the revealed update pattern. w' is randomly chosen from $\mathcal{I}'$. id' is also randomly selected accordingly. Then $\mathcal{S}$ sets $\tau_{upd}' = Enc_{sk_o}(upd')$.

As a result, the environment $\mathcal{Z}$ in Definition 4 cannot distinguish $\tilde{\mathcal{I}}'$, τ_s' and τ_{upd}' from $\tilde{\mathcal{I}}$, τ_s and τ_{upd} in the experiment $\mathbf{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda)$ respectively due to the trusted execution environment enforced by SGX and CPA-secure Enc. ■

Theorem 2: REARGUARD under L_0 is secure against the semi-honest adversary under Definition 4 if the underlying SGX primitives are trusted and encryption is CPA-secure.

Proof: (Sketch). The proof for L_0 construction is similar to that in the L_0^+ model except that

- For search, $\mathcal{S}$ randomly selects a keyword w' in the revealed group from L_0.
- For update, w' is randomly chosen from the revealed group given the group access pattern leakage by L_0.

Thus $\mathcal{Z}$ cannot distinguish $\tilde{\mathcal{I}}'$, τ_s' and τ_{upd}' from $\tilde{\mathcal{I}}$, τ_s and τ_{upd} in the experiment $\mathbf{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda)$ respectively, due to the secure hardware and CPA-secure encryption. ■

In addition, we are also able to realize the security against the malicious adversary by proving the verifiability of the scheme. In general, this can be done through the remote attestation of SGX and replacing the CPA-secure encryption by authenticated encryption. Our design implies *forward privacy* (see Sect. II) as well by hiding the memory trace of index update operation. Moreover, REARGUARD also achieves *query unlinkability* by using semantically secure encryption for the search and update token generation.

Privacy implication of group size in L_0. Attacks on SSE rely on precisely disclosed keyword search and access patterns during the index search phase [9], [10] to uniquely identify the queried keyword and speculate the plaintext dataset information. However, L_0 only leaks the aggregated group search/update pattern, including the number of keywords and length of their associated posting lists in the group. The adversary only knows if the queries are from the same group. The probability of precise keyword-query linkage is $1/n$ for an n-term group. Only given this, the adversary has no advantage in compromising query privacy except for random guessing as we have already proved. Therefore, regardless of the group size, the probability of revealing a query is exactly $1/|\mathbf{W}|$, as same as L_0^+. On the other hand, the adversary may exploit side information, e.g. the lengths of posting lists, context, communication volume, etc., to facilitate query identification. These side-channel attacks are hard to defend even with fully homomorphic encryption [29] and oblivious RAM (ORAM) [24]. The group size makes no differences in this situation.

VI. Implementation and Evaluation

A. Implementation

In order for Intel SGX to offer its strong security guarantees, libraries included in the enclave has to be carefully designed to defend against potential malicious attacks [19]. At the time of writing, SGX SDK (v1.6) supports only C/C++. Many popular search software packages in other programming languages, such as the JAVA-based Lucene, cannot be directly adapted in the programming environment. Further, SGX uses a customized version of C/C++ standard library that only provides a limited subset of functions compared to the standard C library for security reasons. Therefore, even applications written in C/C++ such as Clucene cannot be directly migrated. We developed our own implementation of the search functions under the SGX development environment. To further alleviate index search trace leakage, privacy-sensitive operations are written using the memory-trace oblivious primitives. We built a prototype of REARGUARD with about 4,000 LOC using Intel SGX SDK v1.6 on Intel NUC, running Ubuntu 14.04 TLS. The NUC is powered by Intel i7-6770HQ Skylake CPU with 6MB cache at 2.6 GHz and 8GB DRAM. According to the vendor specification, the read and write speed of the 256GB SSD is 560 MB/s and 400 MB/s respectively. The AES encryption and decryption are implemented with Intel AES-NI instruction. This prototype supports common key query types and functions, i.e. spelling correction, Boolean query, proximity query, phrase query, range query (by converting to Boolean query) and similarity-based ranking.

B. Performance Evaluation

The objective of our evaluation is to measure the performance overhead of the proposed system with elevated security protection. Existing SE work supports only a subset of all the query functions we implemented. Plaintext and SGX-only searches are used as the baselines and compared to the proposed schemes in both L_0 and L_0^+ leakage models. For plaintext search, we evaluate its performance over the index entirely hosted in the main memory. SGX-only search is conducted using SGX protection but without oblivious operations, which can be generally deemed an L_1-SE scheme similar to [17]. Furthermore, we are interested in evaluating the scalability of the proposed system when handling a large-sized index that cannot be completely loaded into the EPC memory.

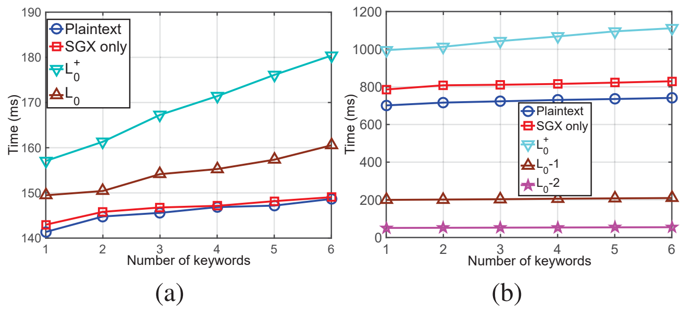


Fig. 6. (a) Search over small-sized index. (b) Scalable search over large-sized index.

The experiments are conducted with a real-world dataset – Enron Email Dataset [30], which contains about half million files and has been extensively employed to evaluate SE schemes [3], [5]. We extracted about $258,000$ keywords and generated a 175MB inverted index after standard term stemming and stop word elimination. We use 80-keyword groups in L_0. The performance was measured over AND Boolean query, similarity-based ranking, and spelling correction (with default $ed = 1$) at the same time in all cases to demonstrate the efficiency and enriched query functions. We selected queries uniformly from the keyword universe. The experimental result is an average of 1,000 trials. For simplicity, we do not consider the optimizations by advanced IR techniques, such as skip pointers for AND Boolean query and index compression, which are compatible with our scheme as we use the same index structure.

Search over small-sized index. We first measure the efficiency of searching over a small-sized index entirely residing inside the enclave. In our experiment, we only have less than 40MB EPC memory available. We randomly select a portion of the original index for all cases. The size of this index is about 35MB consisting of $50,000$ keywords approximately. It is shown in Fig. 6 (a) that time efficiency for all the cases is proportional to the number of keywords in the Boolean queries. We find that search inside enclave is very efficient and only costs about 0.62% additional time for encryption/decryption operations, context switching, etc., compared to plaintext search. On the other hand, REARGUARD are slightly slower than plaintext case due to the proposed oblivious index access functions. Although the L_0^+ design brings an $O(N)$ theoretical complexity, our experiment shows that only $1.16\times$ overhead of plaintext search is incurred. The L_0 construction is faster than L_0^+ as expected and displays a 6% efficiency loss versus plaintext case. Note that 6 or fewer keywords in a query accounts for more than 96% cases in reality [31], therefore the experimental result in Fig. 6 is a representative for the practical use.

Search over large-sized index. For the index with size exceeding the available enclave memory, we divide it into partitions. We set up 5 partition indexes, each about 35MB, in our experiment. Plaintext search is still conducted over the original index in the main memory. The SGX-only search continues loading the index partitions until the match is found. Compared to the plaintext query, the SGX-only case in Fig. 6 (b) shows an average $1.12\times$ efficiency loss due to index loading, encryption/decryption, and context switching. We sequentially load and search over all the partitions for L_0^+, which is only about $1.45\times$ slower than the plaintext case. Although plaintext search time can further speed up by optimizing the index structure, such as using our hierarchical index design, the absolute time cost of L_0^+ is still reasonable considering its strong security assurance. In addition to the overhead caused by SGX, the oblivious index access is the most time-consuming operation.

We split L_0 into two sub-cases. In L_0-1, we exploit a Bloom filter as the first-level index for each second-level index partition. We construct five Bloom filter indexes with all the false positive rate equal to 10^{-20}, which merely consume about 3.03MB EPC memory in total. When a hit is found in the Bloom filter index, we only load and search over the corresponding partition. In L_0-2, we build the Bloom filter for each group index with the same false positive rate as in L_0-1. The total size of the generated first-level index is about 3.3MB. Only the target group index is loaded and searched in the enclave in this case. In Fig. 6 (b), both cases show nearly constant query time. L_0-1 is slightly slower than L_0-2 mainly owing to the relatively large index used there. Because we adopt a hierarchical index structure to allow search over smaller indexes, the two L_0 cases are faster than their competitors. The plaintext and SGX-only search are expected to be more efficient than L_0 with the similar index design.

Index update. Updating index needs extra oblivious write operations compared to the search. Two update query types – *add* and *delete* – on the same index introduce almost the same cost. For an update query toward a group index in L_0, the experiment shows about $1.09\times$ slowdown versus its counterpart search operation while L_0^+ introduces $1.1\times$ time cost of the small-sized index search and $1.08\times$ time cost of the large-sized index search.

VII. Related Work

Search over encrypted data. Curtmola *et al.* [2] proposed the first searchable symmetric encryption scheme in the static setting. It gave two security definitions, i.e. CKA1 and CKA2, where the index search trace leakage is accepted for an efficient query. Kamara *et al.* [3] proposed a dynamic version of [2], supporting file insertion and deletion, but leaking forward privacy during the update. A forward-private SE was first explicitly considered in [4], where an ORAM-related technique was used to alleviate the privacy leakage but incurred non-negligible overhead. Boneh *et al.* [6] built the first public key encryption with keyword search from IBE. All the above SE works only support single keyword query. Recently, the secure Boolean query has been studied in the literature [5] but it still leaks index search trace. Another line of works focus on realizing secure range query. Sun *et al.* [27] solved this problem by reducing a range query to a secure multi-keyword query in the genomic study scenario. Recently a concurrent work by Fuhry *et al.* [17] was proposed to realize secure range query on an SGX-enabled server. However, the security is still consistent with current software-based SSE. In [18], a private database query scheme was proposed also using secure hardware. But it did not provide formal security analysis and their TCB is much larger than ours. Therefore, current SE only covers a small subset of plaintext query functions and cannot provide security guarantees beyond L_1 leakage while maintaining efficiency.

Applications with secure hardware. Recent years has seen increasing interest in building applications on top of secure hardware. Santos *et al.* [13] proposed a trusted language runtime using ARM TrustZone framework to protect the confidentiality and integrity of .NET mobile applications. In IoT setting, Ambrosin *et al.* [14] exploited secure hardware component to enable an asymmetric-key based swarm attestation protocol on IoT devices. There are recent efforts on harnessing Intel SGX to achieve security and privacy preservation for various applications. VC3 [32] was designed to realize the verifiable and confidential execution of MapReduce jobs in an untrusted cloud environment by using SGX. Zhang *et al.* [33] proposed an authenticated data feed system based on SGX, which acted as a trustworthy proxy between HTTPS-enabled servers and smart contracts. Ohrimenko *et al.* [12] studied the problem of multi-party machine learning on an SGX-enabled server. Besides the confidentiality, they also considered the data-dependent memory trace leakage pertaining to the related machine learning algorithms.

VIII. Conclusion

In this work, we propose REARGUARD, the first secure keyword search scheme based on the off-the-shelf trusted hardware to achieve query functions comparable to plaintext IR while ensuring the confidentiality and integrity of the query process. We define two new privacy leakage profiles for SE and present corresponding constructions, which reveal much fewer search footprints than the state-of-the-art software-based solutions. We present approaches beyond the capability of the underlying hardware primitive by designing effective oblivious keyword search functions. Our implementation with the real-world dataset shows its practicality and efficiency.

Acknowledgments

We are grateful to Dr. Ning Zhang for his invaluable suggestions and feedback. This work was supported in part by the NSF grants CNS-1446478, CNS-1405747, and CNS-1443889.

References

[1] D. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. of IEEE S&P*, 2000, pp. 44–55.

[2] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proc. of ACM CCS*, 2006, pp. 79–88.

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

[4] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage." in *NDSS*, vol. 71, 2014, pp. 72–75.

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

[6] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *EUROCRYPT 2004*, pp. 506–522.

[7] CipherCloud, "Cloud data encryption," http://www.ciphercloud.com/technologies/encryption/, 2017.

[8] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *EUROCRYPT 2009*, pp. 224–241.

[9] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. of ACM CCS*, 2015, pp. 668–679.

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

[11] E. Shen, E. Shi, and B. Waters, "Predicate privacy in encryption systems," in *TC*, 2009, pp. 457–473.

[12] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *Proc. of USENIX Security*, 2016, pp. 619–636.

[13] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM TrustZone to build a trusted language runtime for mobile applications," in *Proc. of ASPLOS*, 2014, pp. 67–80.

[14] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter, "SANA: Secure and scalable aggregate network attestation," in *Proc. of ACM CCS*, 2016, pp. 731–742.

[15] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. of IEEE S&P*, 2015, pp. 640–656.

[16] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," *arXiv preprint arXiv:1702.07521*, 2017.

[17] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "HardIDX: Practical and secure index with SGX," in *Proc. of DBSec*, vol. 10359, 2017, p. 386.

[18] S. Bajaj and R. Sion, "TrustedDB: A trusted hardware-based database with privacy and data confidentiality," *IEEE TKDE*, vol. 26, no. 3, pp. 752–765, 2014.

[19] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted rpc interface," in *Proc. of ASPLOS*, 2013, pp. 253–264.

[20] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proc. of ACM HASP*, 2013.

[21] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[22] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press, 2008, vol. 1, no. 1.

[23] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 086, 2016.

[24] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proc. of ACM CCS*, 1987, pp. 182–194.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, no. 6, 2005, pp. 190–200.

[26] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *Proc. of USENIX Security*, 2015, pp. 431–446.

[27] W. Sun, N. Zhang, W. Lou, and Y. T. Hou, "When gene meets cloud: Enabling scalable and efficient range query on encrypted genomic data," in *Proc. of IEEE INFOCOM*, 2017, pp. 1–9.

[28] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li, "Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking," in *Proc. of ACM AsiaCCS*, 2013, pp. 71–82.

[29] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[30] W. W. Cohen, "Enron email dataset," https://www.cs.cmu.edu/∼./enron/.

[31] KeywordDiscovery, "Keyword and search engines statistics," https://www.keyworddiscovery.com/keyword-stats.html?date=2017-04-01.

[32] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Proc. of IEEE S&P*, 2015, pp. 38–54.

[33] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proc. of ACM CCS*, 2016, pp. 270–282.