BISEN: Efficient Boolean Searchable Symmetric Encryption with Verifiability and Minimal Leakage

Bernardo Ferreira[1], Bernardo Portela[2], Tiago Oliveira[2], Guilherme Borges[1],
Henrique Domingos[1] and João Leitão[1]
[1]*NOVA LINCS & DI-FCT-UNL* [2]*HASLab INESC-TEC & DCC-FC-UP*

***Abstract*—The prevalence and availability of cloud infrastructures has made them the *de facto* solution for storing and archiving data, both for organizations and individual users. Nonetheless, the cloud's wide spread adoption is still hindered by dependability and security concerns, particularly in applications with large data collections where efficient search and retrieval services are also major requirements. This leads to an increased tension between security, efficiency, and search expressiveness, which current state of the art solutions try to balance through complex cryptographic protocols that tradeoff efficiency and expressiveness for near optimal security.**

In this paper we tackle this tension by proposing BISEN, a new provably-secure boolean searchable symmetric encryption scheme that improves these three complementary dimensions by exploring the design space of isolation guarantees offered by novel commodity hardware such as Intel SGX, abstracted as Isolated Execution Environments (IEEs). BISEN is the first scheme to enable highly expressive and arbitrarily complex boolean queries, with minimal information leakage regarding performed queries and accessed data, and verifiability regarding fully malicious adversaries. Furthermore, by exploiting trusted hardware and the IEE abstraction, BISEN reduces communication costs between the client and the cloud, boosting query execution performance. Experimental validation and comparison with the state of art shows that BISEN provides better performance with enriched search semantics and security properties.

I. Introduction

Cloud computing has had a profound impact on the way that we design and operate systems and applications. In particular, data storage and archiving is now commonly delegated to cloud infrastructures, both by companies and individual users. Companies typically want to archive large volumes of data, such as e-mails or historical documents, overcoming limitations or lowering costs of their on-premise infrastructures [2], while individual users aim at making their documents easily accessible from multiple devices, or simply avoid consuming storage capacity of their mobile devices [15].

However, data being outsourced to the cloud is often sensitive and should be protected regarding all aspects of dependability. Private information incidents are constant reminders of the growing importance of these issues: governmental agencies impose increasing pressure on cloud companies to disclose users' data and deploy backdoors [24]; cloud providers are responsible, maliciously or accidentally, for critical data disclosures [21]; and even external hackers have gained remote access to users data for limited time windows [28]. Cloud outsourcing services are thus highly incentivized to address these dependability and security requirements. In particular, when storing and updating large volumes of data in the cloud, it is essential to offer efficient, secure, and precise mechanisms to search and retrieve relevant data objects from the archive. This highlights the need for cloud-based systems to balance security, efficiency, and query expressiveness.

To address this tension, Searchable Symmetric Encryption (SSE) [7] has emerged as an important research topic in recent years, allowing one to efficiently search and update an encrypted database within an untrusted cloud server with security guarantees. Efficiency in SSE is achieved by building an encrypted index of the database and storing it in the cloud [17]. At search time, a cryptographic token specific to the query is used to access the index, and retrieved index entries are decrypted and processed. To minimize communication overhead, most SSE schemes delegate the execution of cryptographic computations to the cloud, as multiple index entries would otherwise have to be requested and downloaded to the client. However, performing sensitive operations in the cloud also leads to significant information leakage, including the leakage of document identifiers matching a query, the repetition of queries, and the compromise of forward and backward privacy [34] (respectively, if new update operations match contents with previously issued queries, and if queries return previously deleted documents). These are common, yet severe, flavors of information leakage that pave the way for strong attacks on SSE, including devastating file-injection attacks [38]. Another relevant limitation of SSE schemes is query expressiveness, as most solutions only support single keyword match [13] or limited boolean queries (e.g., forcing queries to be in Conjunctive Normal Form and not supporting negations) [25]. This hinders system usability and may force users to perform multiple queries in order to retrieve relevant results, leading to extra communication steps and additional information leakage.

In this paper we address these limitations by presenting BISEN (Boolean Isolated Searchable symmetric ENcryption), a new provably-secure boolean SSE scheme that improves query expressiveness by supporting arbitrarily complex boolean queries with combinations of conjunctions, disjunctions, and negations (e.g., "(cancer $\vee$ terminal) $\wedge \neg$ cirrosis"). This is a significant improvement over the current state of the art, since supporting boolean queries is fundamentally more challenging than single-keyword queries and addressing negations is a non trivial task. Furthermore, BISEN also boosts performance by minimizing the number of communication

steps and amount of data transferred between clients and cloud servers. A central insight in the design of BISEN is the fact that we can securely delegate critical computations to the cloud by leveraging on a hybrid solution that combines standard symmetric-key cryptographic primitives (e.g., Pseudo-Random Functions and Block-Ciphers [26]) with remote attestation capabilities offered by modern trusted hardware, formally captured by an abstraction called Isolated Execution Environments (IEEs) [4].

An IEE is an environment that allows applications to execute in isolation from all external interference (including co-located software and even a potentially malicious Hypervisor/OS) and that provides a mechanism for the remote attestation of computed outputs. Until recently, such an abstraction could only be built through hardware that was unfeasible to deploy in cloud infrastructures, however recent advances in trusted computing have made IEEs available in commodity hardware. Prominent examples include Intel SGX [16] and ARM TrustZone [1], which are being deployed in current desktop and mobile processors and will soon become available as part of many cloud infrastructures [32].

A main advantage of designing our system to leverage the IEE abstraction lies in its portability, as our solution can be easily instantiated using different existing (or future) IEE-enabling technologies as they become available in cloud platforms, while preserving security guarantees. This is also relevant considering recent attacks on trusted hardware [35], [36]. To further increase this portability, we extend the IEE formalization to support very lightweight hardware technologies (such as Intel SGX, with its limited Enclave Page Cache size of 128MB), complemented with SSE techniques and cryptographically protected accesses to more abundant untrusted resources in the machine hosting the IEE or in other external cloud storage services. This approach has mutual benefits: SSE techniques allow extending IEE trusted resources beyond hardware limitations in a secure way, minimizing assumptions regarding the underlying technology employed; and IEEs allow increasing the performance, scalability, and security of SSE schemes.

In summary, in this paper we provide the following main contributions:

- We propose and formalize an approach for extending trusted hardware resources, integrating it in the IEE abstraction of Barbosa et al. [4]. This approach allows IEEs to support and operate on very large databases, and may be of particular interest for other applications;
- We design BISEN, a Boolean SSE scheme based on the previous approach and that can support all Boolean operands and formulas. By leveraging IEEs as remote trust anchors, BISEN is able to move most client-side computations to the server, providing overall reduced computation, storage, and communication overheads. BISEN provides verifiability against fully malicious adversaries, supports dynamic updates with forward and backward privacy, and queries only reveal which encrypted index entries are accessed;
- We implement a prototype of BISEN based on Intel SGX, which we run on real world datasets to experimentally validate its efficiency properties. Our prototype is open-source and available at *https://github.com/sgtpepperpt/BISEN*.

II. Background and Related Work

Isolated Execution Environments (IEEs) From a high level, and as defined by Barbosa et al. [4], an IEE is an idealized random access machine, running a fixed program, and whose behaviour can only be influenced by a well-specified interface that allows input/output interactions with the program. Isolation guarantees in IEEs follow from the requirements that: the I/O behaviour of programs running within them can only depend on themselves, on the semantics of their language, and on inputs received; and that the only information revealed about these programs must be contained in their I/O behaviour. This abstraction allows for the formal treatment of remote attestation mechanisms offered by technologies such as SGX and TrustZone, which were shown in [4] to be sufficient for the deployment of an Outsourced Computation protocol.

Building on these definitions, Bahmani et al. [3] demonstrated how to refine the IEE attestation mechanism to enable for the deployment of *general multiparty computation*. Their design follows two main stages. First, clients leverage remote attestation mechanisms to perform a key exchange agreement with the IEE and establish a secure communication channel. Afterwards, clients use these channels to interact with a reactive functionality on the IEE, exchanging encrypted inputs and outputs with confidentiality and integrity guarantees. The usage of sequence numbers in communications made through these channels also prevents a malicious server from repeating requests. In this work we will leverage on the IEE abstraction and this protocol, further extending it by allowing the IEE to interact with untrusted storage resources with privacy, integrity, and verifiability guarantees.

Searchable Symmetric Encryption (SSE) SSE deals with the problem of how to efficiently search and update an encrypted database [17]. To achieve this goal, SSE schemes usually build an encrypted index of the database (e.g., an inverted list index [31]), hence reducing the previous problem to the easier one of searching and updating an encrypted index. This approach has additional advantages, as the index allows search performance to be sub-linear on the database size, and the data itself can eventually be stored on a second storage system with different security guarantees. In its most simple version, keys of this encrypted index are message authentication codes (MACs) of keywords, and values are symmetrically encrypted versions of document identifiers.

To search the encrypted index, the client transforms his query into a cryptographic token (usually composed of a pair of cryptographic keys), which is used by the server to find and decrypt relevant index entries. This approach allows the encrypted index to provide zero-leakage when it is on the rest, however some information patterns must be leaked when it is updated or queried, as a necessary tradeoff for achieving practical performance (also providing zero-leakage during computation would require expensive techniques such

as Oblivious-RAM [23]). Information leaked by SSE schemes includes search patterns (if a query has occurred in the past and when) and access patterns (which index entries are accessed by the query). Query-recovery attacks have been demonstrated based on both patterns [12], [30], although requiring large a-priori database knowledge (around 90%) or the adversarial ability to inject files [38].

Forward and backward privacy are also important security definitions in SSE [34]. Forward privacy enforces that update operations should not reveal anything regarding updated keywords, even if combined with previously issued query tokens [8], and helps partially mitigating file-injection attacks. Backward privacy enforces that search operations should only reflect the current state of the database and should reveal nothing regarding deleted keywords [9].

SSE schemes usually only support single keyword queries, as supporting boolean multi-keyword queries with similar security guarantees and performance is a fundamentally more difficult problem [14], [25]. Even the most recent Boolean SSE scheme to date [25] still provides: limited usability, as it does not support negations and queries must be in Conjunctive Normal Form (CNF), possibly forcing users to rewrite their queries; limited performance, requiring quadratic server storage in the number of unique keywords in the database and exhibiting quadratic search performance in the query size; and limited security, as queries leak the search and access patterns of some of their individual keywords and of the resulting conjunctions/disjunctions. This is due to the difficulty of managing complex multi-map data-structures required by the authors for supporting boolean queries, which we show in this work to be avoidable by leveraging remote trust anchors expanded with cryptographically secured accesses to large untrusted storage.

Cryptographic Protocols based on Trusted Hardware Recent works have demonstrated the benefits that trusted hardware like Intel SGX can bring to the design of cryptographic protocols. Iron [20] used Intel SGX to develop a practical Functional Encryption (FE) scheme [20]. SSE, as most schemes for privacy-preserving computations, can be seen as specialization of FE, meaning that the approach proposed by the authors could also be employed to solve the problems we address in this work. However, our approach is specifically tailored for solving the challenges posed by searching encrypted data, optimizing performance and efficiency as no general purpose approach traditionally can.

ZeroTrace [33] provided a more efficient protocol for Oblivious-RAM based on SGX. Their techniques can be used to complement our approach, as a way to further reduce information leakage and provide further resilience to side-channel attacks [36]. HardIDX [22] used SGX for efficiently supporting range queries in SSE. Their approach has similarities with ours, but its focus is on a fundamentally different problem (range queries). Additionally, it only supported static databases and required the client to build the encrypted index. In contrast, or work supports dynamic updates with minimal leakage and moves most computations to the cloud in a secure way, both considered essential for practical applications.

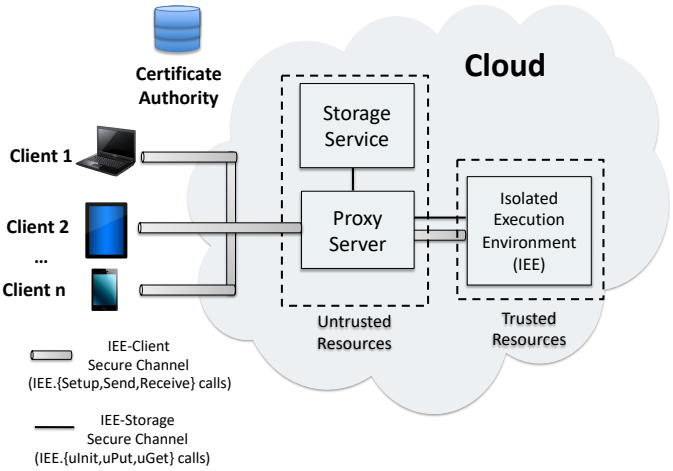

Fig. 1: Overview of the proposed approach.

III. Technical Overview

The main idea of BISEN is for clients to leverage IEEs as remote trust-anchors within the cloud, supporting efficient update and search operations on their cloud-stored encrypted databases. However, it would be unfeasible to maintain a whole database index within a resource-restricted IEE. As such, our proposal is to leverage a highly efficient environment for computations (the IEE) and a virtually infinite source for external storage (the cloud). BISEN combines these tools in its IEE-sided code, processing queries within its isolated memory, and relying on an cloud service for storing encrypted data.

Figure 1 provides an overview of BISEN's architecture and its components. BISEN starts with a bootstrapping phase, where a client contacts a cloud server to initiate the IEE with BISEN's code. We call this server the Proxy Server, as it operates the IEE, manages all of its communications, and orders concurrent accesses. When started, the IEE initiates its state and asks a Cloud Storage Service to create BISEN's (initially empty) encrypted index. This storage service will basically be responsible for large-scale storage, and can even be instantiated through pure storage solutions (e.g. AWS S3), as it only needs to support put/get operations. Hardcoded in BISEN's code is the public key of a trusted Certificate Authority, meaning that the IEE will only accept communications from clients that present a valid certificate signed by it.

After bootstrap, clients can contact the proxy server to remotely attest it created an IEE with BISEN's code and to establish secure communication channels with it. Secure channels are established through a key-exchange algorithm based on remote attestation and the clients' public keys, as in [3]. Through these channels, clients can issue update and search operations to the IEE, which it processes by contacting the storage service and accessing BISEN's index.

Updates allow both adding or removing keywords to/from documents. In either case, a new encrypted entry is added to BISEN's index, where its key is composed of a deterministic cryptographic token uniquely combining the keyword and

document, and its value is an encrypted message that includes the document id and a flag indicating if the operation is an addition or removal. This approach guarantees that both operations are indistinguishable, a necessary condition for preserving forward and backward privacy.

Search operations take a boolean query as input. The IEE processes this query, retrieves and decrypts relevant index entries from the storage service, calculates the resulting set of document ids, and finally returns this set to the client.

Adversary Model. The clients, the IEE, and the Certificate Authority are the only trusted participants in BISEN. The Proxy Server and Storage Service are considered fully malicious, i.e., they may attempt to break data privacy, integrity, or computation correctness. Networking channels are also considered untrusted. Denial of service is considered out of scope for this work, as the cloud controls the whole infrastructure, but may be addressed in future works through cross-cloud replication.

Application Scenario. An interesting application scenario for BISEN is that of encrypted archival of email in the cloud. In such a scenario, users would be able to securely outsource the storage and management of their emails to a third-party cloud provider, while still being able to have rich search features that are commonly found in todays unsecured email cloud archival services. As studied by Zheng et al. [38], cloud email is an example scenario that can be easily targeted by file-injection attacks, hence this application enforces the need to improve the security of SSE schemes to withstand fully malicious adversaries. Furthermore, preserving forward privacy is known to help mitigate such attacks [38], and backward privacy may have important implications in future attacks as well [9]. Overall, minimizing information leakage should be a top priority when deploying SSE schemes in practical scenarios.

IV. BISEN

In this section we present BISEN's full details. We start with some required notations and definitions (§ IV-A), then we present BISEN's protocols (§ IV-B), and finally we analyse its security (§ IV-C).

A. Notations and Definitions

General Notations. In this paper we denote by λ the security parameter and $\mu(\lambda)$ a negligible function in it. We will use the standard security notions of variable-input-length Pseudo-Random Functions (PRF, instantiated as an HMAC in our implementation) [5] and authenticated encryption schemes ensuring *indistinguishability under chosen-ciphertext attacks* (IND-CCA) [26]. We assume the keys of these primitives to be uniformly sampled from $\{0,1\}^\lambda$ by the key generation algorithm. We consider adversaries to be probabilistic algorithms, running in time polynomial on security parameter λ.

IEE Notations. IEE behavior and interactions with clients can be abstracted as $\mathsf{IEE} = (\mathsf{Setup}, \mathsf{Send}, \mathsf{Receive})$, as follows:

- $\mathsf{IEE.Setup}(1^\lambda, \mathsf{pk}_c)$ corresponds to IEE bootstrapping (if it hasn't been initialized yet) and secure channel establishment. Setup takes security parameter 1^λ and client public key pk_c as input, and produces state $\mathsf{st}_{\mathsf{IEE}}$ with the exchanged key, if pk_c could be verified.
- $\mathsf{IEE.Send}(\mathsf{st}_{\mathsf{IEE}}, \mathsf{m})$ can be used by the client or IEE, and uses the secure channel established by Setup to encrypt m with the key in $\mathsf{st}_{\mathsf{IEE}}$. This outputs c and updates state $\mathsf{st}_{\mathsf{IEE}}$.
- $\mathsf{IEE.Receive}(\mathsf{st}_{\mathsf{IEE}}, \mathsf{c})$ uses the channel to retrieve encrypted message c using the key in $\mathsf{st}_{\mathsf{IEE}}$. This outputs m and updates state $\mathsf{st}_{\mathsf{IEE}}$.

Additionally, and differently from the IEEs original specification [3], [4], we consider IEEs to rely not only on *trusted* state, which is assumed to be incorruptible by the underlying system, but also on *untrusted* state (represented in BISEN through the Storage Service), which has to be explicitly protected through cryptographic algorithms. To establish interactions with this *untrusted* state, and following a dictionary-like notation, we define three new calls in the IEE abstraction:

- $\mathsf{IEE.uInit}()$ initializes an empty data-structure D in untrusted storage outside the IEE. It outputs D, making it available for future uPut and uGet operations;
- $\mathsf{IEE.uPut}(D, \{l_i, v_i\}_{i=0}^*)$ accesses untrusted storage and stores a group of entries $\{l_i, v_i\}_{i=0}^*$ in data-structure D. It outputs updated structure D.
- $\mathsf{IEE.uGet}(D, \{l_i\}_{i=0}^*)$ accesses untrusted storage and outputs a group of values $\{v_i\}_{i=0}^*$, stored in positions $\{l_i\}_{i=0}^*$ of data-structure D.

Formally, we consider uInit and uPut to additionally produce an execution trace, containing the operation, its input, and the output. In the security experiment this trace is given directly to the adversary, capturing the notion that all data stored through this mechanism is considered leakage. Since we are considering a fully malicious adversary, all values returned by uGet can be set by the adversary.

SSE Notations. In SSE, a database DB is composed by a collection of d documents, each with a unique identifier id and containing a set of keywords W. For a keyword w, $\mathsf{DB}(w)$ is the set of documents where it occurs. The total number of document/keyword pairs is denoted by n and is stored in an encrypted index I, which is a dictionary structure mapping each unique keyword w to a list of matching documents $(\mathsf{id}_0, .., \mathsf{id}_{|\mathsf{DB}(w)|-1})$ and allowing queries to be performed in time sub-linear in n. $\phi(\bar{w})$ is a boolean query composed of a set of keywords $\bar{w}$ and satisfying a boolean formula ϕ. $\mathsf{DB}(\phi(\bar{w}))$ represents the set of documents satisfying $\phi(\bar{w})$.

A *multi-client dynamic boolean searchable symmetric encryption scheme* $\prod = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ consists of three protocols between a client and a server (in our case, the IEE):

- $\prod.\mathsf{Setup}\ (1^\lambda, \mathsf{pk}_c)$ starts the scheme, with inputs security parameter 1^λ and client public key pk_c. At the end of the protocol the client has secret parameter K and, if the scheme

hadn't been initialized yet by another client, the server has the (initially empty) encrypted index I.

• $\prod$.Update $(\mathsf{K}, \mathsf{op}, w, \mathsf{id})$ updates the database with inputs secret parameter K, operation $\mathsf{op} = \{\mathsf{add}, \mathsf{del}\}$ (i.e., an addition or deletion of a keyword), keyword w, and document identifier id. In the end, the server outputs updated I.

• $\prod$.Search $(\mathsf{K}, \phi(\bar{w}))$ queries the database with inputs K and boolean query $\phi(\bar{w})$. In the end, the client outputs a set of document identifiers. If for any possible inputs this output is $\mathsf{DB}(\phi(\bar{w}))$, we say that $\prod$ is *correct*.

B. The Scheme

Figure 2 details BISEN's protocols. In the Setup algorithm, the client starts by contacting the proxy server and invoking the IEE.Setup protocol. If the IEE hasn't been initiated yet, the proxy server starts it, and then redirects the client's setup message to it. The IEE starts by performing a key-exchange algorithm with the client to create the secure channel between the two. From this algorithm results state $\mathsf{st}_{\mathsf{IEE}}$, which contains the corresponding communication key. Then, if this is its first execution, the IEE initiates its state: a key k_F ,to be used with PRF F; key k_E, to be used with encryption scheme Θ; dictionary of counters W, mapping each unique keyword w in the database to an integer counter c initialized at zero (each increment in c represents a new document containing w); and counter nDocs, which counts the number of unique database documents and will be used in the Search protocol to help resolve Boolean queries with negations. Finally, the IEE also asks the storage service to initiate index I, through the IEE.uInit call.

The Update protocol can be used both for adding or deleting keywords to/from documents, depending only on the value of input op. Moreover, the protocol follows the same specification for both, and for adverseries, they are indistinguishable. When performing an update, the client starts by sending $(\mathsf{op}, \mathsf{id}, h_w)$ to the IEE through their secure channel (IEE.$\{$Send, Receive$\}$ calls), where h_w is a hash of keyword w, performed to normalize keyword lengths and make updates for different keywords indistinguishable. Upon receiving this message, the IEE builds l, the label (or index key) for this update, by first applying F on h_w and k_F to produce k_w, and then on k_w and c (the keyword's counter, retrieved from W). This double application of F allows merging both h_w and c with a secret parameter (k_F), producing a secure label that is unique for this update. l determines the position in index I where the update will be stored. As index value, the IEE encrypts $(l, \mathsf{op}, \mathsf{id})$ with Θ, an authenticated encryption scheme. Θ ensures the preservation of both privacy and integrity of encrypted index values. Furthermore, by including l in the encrypted index value, the IEE can validate during Search operations that the server is returning correct responses when it requests index values from untrusted storage. Finally, the IEE sends this new index entry to the server for storage, through the IEE.uPut call, and increments nDocs if this is a new document.

When searching with a boolean query, the client also sends h_w to the IEE, for each keyword w in the query. Additionally the client sends ϕ, the boolean formula of the query that the IEE needs for computing search results. Given this message, the IEE recalculates all labels for the inputed keywords (as in Update), requesting the respective index positions from the server through IEE.uGet. To hide any possible patterns in the query structure, the IEE randomly permuts label order and requests all at once (or alternatively in single, but successive requests, if the storage service does not provide such semantics). The IEE proceeds to decrypt these entries with Θ, verifies if the server returned correct index values for each label (aborting the protocol otherwise), and resolves the boolean query by applying ϕ to the results.

In this setting, the process of resolving a boolean query can be described in light of set operations. Searching for a keyword results in a set of document identifiers. When two or more keywords are queried, their sets can be unionized or intersected, depending if ϕ specifies disjunctions or conjunctions between them, respectively. For queries of three or more keywords, parentheses can also be used to specify precedence between boolean operands. Performing negations is somewhat more complicated however, since inverting sets implies having knowledge of the range of all possible values (in this case, all document ids). To circumvent this issue we define that documents are identified by the incremental values of counter nDocs, starting at zero. Additionally, correctness of document identifiers is assured by enforcing that the ids inputed on Update belong to the range $[0..\mathsf{nDocs}+1]$. Using this approach, the system can easily filter results for all existing documents, and thus efficiently support negations by searching for a keyword and inverting its document set[1].

Optimizations and Extensions. An important goal in BISEN is being able to support lightweight IEE technologies, such as Intel SGX with its restricted EPC size of 128MB. The proposal to extend IEE storage with cryptographically secured accesses to untrusted storage partially supports this goal. However, when performing a search in very large databases, intermediary metadata that the IEE needs to process may still be too large for such hardware restrictions. In these scenarios, incremental computing principles can be applied to ensure scalability: the IEE can dynamically calculate how many index entries will fit in its limited trusted storage, request that many entries through the IEE.uGet call, process and discard them, preserving only partial search results and merging them with the results of previous iterations of this algorithm.

Increasing the number of IEEs is also a useful extension, allowing BISEN to scale regarding both database size and client concurrency. The best way to achieve this is to make IEEs stateless: clients could generate keys k_E and k_F and share them with new IEEs (which would also help with possible IEE termination issues by the proxy server), while remaining state (W and nDocs) could be stored in the storage service in a secure way and with a concurrency control mechanism.

[1]We assume that ids are never effectively removed, i.e., even if a document has all of its keywords deleted, its id will still exist and will represent an empty document. This approach has other benefits as well, including the possibility of recycling document ids.

$\underline{\mathsf{Setup}(1^\lambda, \mathsf{pk}_c)}$

Client:

$\mathsf{st}_{\mathsf{IEE}} \leftarrow_\$ \mathsf{IEE.Setup}(1^\lambda, \mathsf{pk}_c)$

Server:

$\mathsf{IEE.Setup}(1^\lambda, \mathsf{pk}_c)$

IEE:

$\mathsf{st}_{\mathsf{IEE}} \leftarrow_\$ \mathsf{IEE.Setup}(1^\lambda, \mathsf{pk}_c)$
$\mathsf{I} \leftarrow \mathsf{IEE.uInit}()$
$\mathsf{k}_F \leftarrow_\$ \mathsf{F.Gen}(1^\lambda)$; $\mathsf{k}_E \leftarrow_\$ \Theta.\mathsf{Gen}(1^\lambda)$
$\mathsf{W} \leftarrow \mathsf{Init}()$; $\mathsf{nDocs} \leftarrow 0$

$\underline{\mathsf{Update}(\mathsf{op}, w, \mathsf{id})}$

Client:

$h_w \leftarrow \mathsf{H}(w)$
$m^* \leftarrow_\$ \mathsf{IEE.Send}(\mathsf{st}_{\mathsf{IEE}}, \{\mathsf{op}, \mathsf{id}, h_w\})$
Send m^* to Server.

Server:

Send m^* to IEE.

IEE:

$\{\mathsf{op}, \mathsf{id}, h_w\} \leftarrow \mathsf{IEE.Receive}(\mathsf{st}_{\mathsf{IEE}}, m^*)$
$c \leftarrow \mathsf{Get}(\mathsf{W}, h_w)$
if $c = \perp$ **then**
    $c \leftarrow 0$
else
    $c \leftarrow c + 1$
$\mathsf{W} \leftarrow \mathsf{Put}(\mathsf{W}, h_w, c)$
$\mathsf{k}_w \leftarrow \mathsf{F.Run}(\mathsf{k}_F, h_w)$; $l \leftarrow \mathsf{F.Run}(\mathsf{k}_w, c)$;
$\mathsf{id}^* \leftarrow_\$ \Theta.\mathsf{Enc}(\mathsf{k}_E, (l, \mathsf{op}, \mathsf{id}))$
$\mathsf{I} \leftarrow \mathsf{IEE.uPut}(\mathsf{I}, l, \mathsf{id}^*)$
if $\mathsf{id} > \mathsf{nDocs}$ **then**
    nDocs++

$\underline{\mathsf{Search}(q)}$

Client:

$\{\bar{w}, \phi\} \leftarrow \mathsf{ProcessBooleanQuery}(q)$; $C \leftarrow [\,]$
for all $w \in \bar{w}$ **do**
    $h_w \leftarrow \mathsf{H}(w)$; $C \leftarrow h_w : C$
$m^* \leftarrow_\$ \mathsf{IEE.Send}(\mathsf{st}_{\mathsf{IEE}}, \{C, \phi\})$
Send m^* to Server.

Server:

Send m^* to IEE.

IEE:

$\{C, \phi\} \leftarrow \mathsf{IEE.Receive}(\mathsf{st}_{\mathsf{IEE}}, m^*)$; $Q \leftarrow \mathsf{Init}()$
for all $\mathsf{H}_w \in C$ **do**
    $\mathsf{k}_w \leftarrow \mathsf{F.Run}(\mathsf{k}_F, h_w)$
    $c \leftarrow \mathsf{Get}(\mathsf{W}, h_w)$; $L \leftarrow [\,]$
    for all $c_i \leftarrow 0 \ldots c$ **do**
        $l \leftarrow \mathsf{F.Run}(\mathsf{k}_w, c_i)$; $L \leftarrow l : L$
    $Q \leftarrow \mathsf{Put}(Q, \mathsf{k}_w, L)$
$L' \leftarrow \mathsf{Flatten}(Q)$
$\Pi \leftarrow_\$ \mathsf{RandomPermutation}(1^\lambda)$; $L' \leftarrow \Pi(L')$
$D' \leftarrow \mathsf{IEE.uGet}(\mathsf{I}, L')$; $D \leftarrow [\,]$
for all $\mathsf{id}^* \in D'$; $l' \in \mathcal{L}'$ **do**
    $(l, \mathsf{op}, \mathsf{id}) \leftarrow \Theta.\mathsf{Dec}(\mathsf{k}_E, \mathsf{id}^*)$; $\mathsf{Verify}(l, l')$
    $D \leftarrow \{\mathsf{op}, \mathsf{id}\} : D$
$D \leftarrow \Pi^{-1}(D)$; $Q' \leftarrow \mathsf{Join}(Q, D)$
$R \leftarrow \mathsf{Resolve}(\phi, Q', \mathsf{nDocs})$; $r^* \leftarrow_\$ \mathsf{IEE.Send}(\mathsf{st}_{\mathsf{IEE}}, R)$
Send r^* to Server.

Server:

Send r^* to Client.

Client:

$R \leftarrow \mathsf{IEE.Receive}(\mathsf{st}_{\mathsf{IEE}}, r^*)$

Fig. 2: Our BISEN scheme based on $\mathsf{IEE} = (\mathsf{Setup}, \mathsf{Send}, \mathsf{Receive}, \mathsf{uInit}, \mathsf{uPut}, \mathsf{uGet})$, PRF $\mathsf{F} = (\mathsf{Gen}, \mathsf{Run})$, authenticated encryption scheme $\Theta = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, and hash function H.

Encrypted index I is already monotonically crescent, hence avoiding concurrency issues.

Another design choice is how to perform delete operations. In the original approach by Cash et al. [13], delete tokens should be stored in a separate set D, indexed by a PRF over both the deleted keyword and document id. This approach has the advantage of only storing one index entry per keyword/document deletion, while BISEN stores a new entry for each deletion submitted by the client (even if repeated). However it makes keyword deletions distinguishable from additions, and requires contacting the server twice, first for accessing index I and then for accessing D.

C. Security Analysis

In BISEN our goal is for Update operations to have no leakage and Search operations to only reveal message lengths and which encrypted index entries are accessed, which we abstract as labels. This is similar to the access pattern of previous SSE schemes, however it captures a stronger security notion since document identifiers are protected at all times. Moreover, executing Search for two distinct queries can leak the same label set, thus reducing the adversarial ability to distinguish between queries. For instance, boolean formulas $\phi_1 = w_1 \vee w_2$ and $\phi_2 = w_1 \wedge w_2$, although representing different queries, access the same label set.

Formally, BISEN's security is parametrized by three leakage functions $(\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Update}}, \mathcal{L}_{\mathsf{Search}})$. From these, only $\mathcal{L}_{\mathsf{Search}}$ produces leakage, detailed as follows:

$$\mathcal{L}_{\mathsf{Search}}(\mathsf{q}) = ((|\phi| + N), |\mathsf{Resolve}(\phi, \mathsf{D}, \mathsf{nDocs})|, \mathsf{L})$$

where the first part corresponds to the length of the input message ($|\phi|$ is the length of the boolean formula of the query and N is the number of distinct keywords in it), the second part is the length of the query response, and L is the set of labels relevant for the resolution of the query.

The Update protocol has no leakage as all input and output messages are of equal length and cryptographic operations are performed inside the IEE. Having updates with zero leakage also ensures forward privacy [8]. Backward privacy (specifically, backward privacy with update pattern [9]) is ensured by storing additions and deletions in an indistinguishable fashion and filtering results in the IEE.

Moreover, the employed cryptographic mechanisms ensure security as follows: prf-security and uniqueness in keywords and counters ensures indistinguishability of labels from outputs from a random function applied to a unique counter; unforgeability of Θ ensures that adversaries cannot produce a

ciphertext that does not exactly match the stored data for said keyword/counter pair on the corresponding update request; the security of the IEE channel and the sequence numbers used prevent adversaries from emulating a fake BISEN execution, forging client requests, altering the order of messages exchanged, or performing replay attacks. In our companion technical report [19] we provide a formal security proof of these statements in the real/ideal standard cryptographic model [26].

V. Implementation

We implemented a prototype of BISEN in C/C++, with around 6200 lines of code. Our prototype is based on Intel SGX [16], using its remote-attestation and enclave management primitives to provide the IEE functionalities required by BISEN. To bootstrap the IEE and establish secure Client-IEE channels, we leveraged the SGX-based open-source implementation of Bahmani et al. [3], adapting it for BISEN. Their original implementation employed the NaCl cryptographic library [6] for elliptic curve algorithms and other cryptographic primitives. Our adapted implementation relies on LibSodium [29] instead, which is a more complete and up-to-date constant-time cryptographic library, partly based on AES-NI. Constant-time cryptographic algorithms based on hardware implementations and oblivious primitives allow us to prevent side-channel leakage of the SGX enclave, including page and cache level leakage.[2]

We instantiate PRF F with LibSodium's SHA256-HMAC implementation, and Θ with its authenticated encryption algorithm, XSalsa20 stream cipher with Poly1305 MACs. Since LibSodium is not ready for SGX deployment, we prepared an SGX-compatible version by (among other steps) removing all unsupported functions in SGX and replacing randomness functions with their equivalents from Intel's RNG library.

Regarding attestation, the employed mechanism follows the design originally proposed in [4], where each program running on an IEE must produce a signature of its code and I/O trace thus far. For Intel SGX, this relies on the Quoting enclave, which uses the EPID group signature scheme [11] to produce a signature (quote) binding the enclave execution trace with the code that produced such trace. Verification of quotes is performed by the client through Intel's Attestation Service.

For the IEE to interact with the encrypted index I, we leveraged on SGX ocalls. Additionally, concurrent accessed are managed by leveraging enclave multi-threading and by using concurrent data structures that only block on writes. Our implementation is open-source and available at: *https://github.com/sgtpepperpt/BISEN*.

VI. Experimental Evaluation

We now experimentally evaluate BISEN, using the prototype implementation described in the previous section.

[2]Note that these countermeasures are not sufficient to protect from devastating attacks such as [35]. This is where BISEN benefits from relying on the IEE abstraction, as one can instead implement it on hardware resilient to speculative execution attacks if this is a realistic concern, e.g. MI6 [10].

Experimental Test-Bench. We present performance results for BISEN and its Search and Update protocols. As IEE and proxy server, we used an Intel NUC i3-7100U with built-in SGX support, 2.4GHz of CPU frequency, 8GB of RAM, 256GB of SSD storage, running Ubuntu Server 18.04.1. As storage service we used a server with an AMD Opteron 6272 CPU with 64GB of RAM. Both machines were deployed on a one gigabit ethernet network. To evaluate the impact of remote communications, and since we already had the previous hardware available, we leveraged the cloud to deploy the client instead, using an AWS EC3 t3.large instance. The round-trip time between client and proxy server was 41.377ms and the max transmission rate was 50Mb/s. As dataset, we used a the english wikipedia dump of August 2018 [37] with around 60GB of uncompressed text data, 5.5 million documents, and 464 million keyword/document pairs. Measurements are based on an average of 50 independent executions.

Experimental Evaluation Roadmap. The goal of our experimental work is to answer the following questions: $i.$) what are the storage costs of BISEN; $ii.$) what is the performance cost (i.e., total time consumed) to process and store a whole dataset through a batch of Update protocol invocations, and how does this performance evolve as we scale the dataset's size; $iii.$) what is the performance cost of executing different types of Search queries, including queries with multiple conjunctions, disjunctions, and negations, considering different database sizes, the selectivity of queried keywords (i.e., the size of returned results), and the query size; and $iv.$) how does BISEN's performance compare with the state of art in Boolean SSE, namely the recent IEX-2LEV scheme [25].

A. Storage Costs

In BISEN, clients only store one cryptographic key (32 bytes), which is used for secure communication with the IEE. The IEE also stores this key, plus k_F and k_E ($3 * 32 = 96$ bytes). Additionally it stores dictionary of counters W, which keeps a counter (4 bytes) and a hash (32 bytes) per entry, with one entry per unique keyword in the database. For the English Oxford dictionary containing 616500 unique word-forms, this results in an upper bound of around 20MB IEE storage, while e-mail date searches for individual days over 200 years could correspond to around 3MB IEE storage. The storage service stores index I, which can grow due to the security guarantees provided (83 bytes per entry), nonetheless with cloud storage this can be more seamlessly scaled.

B. Update Performance

Figure 3 reports the performance results for the Update protocol of BISEN. The y-axis represents time elapsed (in seconds), while the x-axis represents the update size in terms of keyword-document pairs (i.e., how many entries are being added to index I at once, with a single batch of multiple Update protocol invocations). Results were measured at different batch update sizes (up to 464 million pairs) and are reported for networking and for the three main protocol executors in separate, namely the client, IEE, and storage

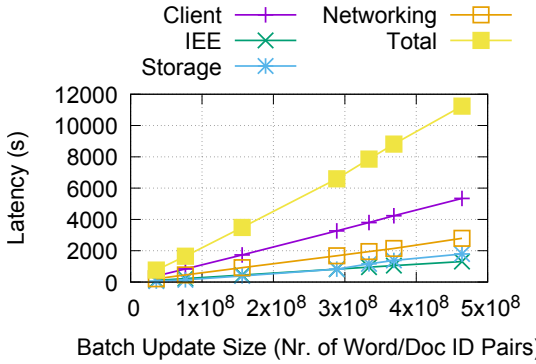

Fig. 3: Performance of the Update protocol.

service. Proxy performance is omitted for simplicity, as it only forwards messages and its execution is highly efficient. Total results are also reported for convenience of the reader.

Analysing the obtained results, one can conclude that BISEN's performance scales linearly with the size of the batch update (Total line in Figure 3). An update for a single document with 640 keywords takes 29ms, while a batch update of multiple documents totaling 464 million keywords takes 11.239 seconds (around 3 hours). This means that performance of Update invocations is mostly unaffected by the current database size. This is a natural observation, since this protocol does not depend on previous operations. These results also reflect the good performance properties of modern trusted hardware technologies, namely Intel SGX. The results for network performance basically show the cost of uploading data to the cloud, as BISEN adds very little cryptographic expansion: communications are encrypted with standard symmetric-key cryptography, and keywords are only hashed.

Regarding the performance of each protocol participant in separate, we can observe that time spent in the IEE and Storage Service is roughly similar, with a tendency for the Storage to become a bottleneck for larger operations. While we consider a single storage server, distributing this service across multiple machines might mitigate its weight in the operation. In turn, the IEE is responsible for simple cryptographic computations, entering enclave mode in SGX, and exiting this mode to store data through SGX ocalls, which is reflected in the latency for the maximum update (1300 seconds for 464 million keyword/document pairs). The largest slice of processing is on the client, which seems contradictory as from BISEN's specification (Figure 2), the client performs very few computations. From our analysis, we argue that these results are due to necessary pre-processing: the client has to process the whole dataset from disk, parsing its keywords, stemming them and filtering stop-words [31]. The introduction of parallel processing on our client prototype can help improve these results. Additionally, in applications where documents are created and edited online for instance, this overhead would be greatly reduced.

C. Search Performance

To analyse the performance of the Search protocol, we conducted experiments with different types of queries, measuring in all cases how performance scaled with the increase in database size. For transparency in evaluation, in the following experiments we used the most popular keywords in the english language, i.e., the keywords that appear in more documents (also known as having high selectivity). From first to twelve, these are: *time*, *person*, *year*, *way*, *day*, *thing*, *man*, *world*, *life*, *hand*, *part*, and *child*

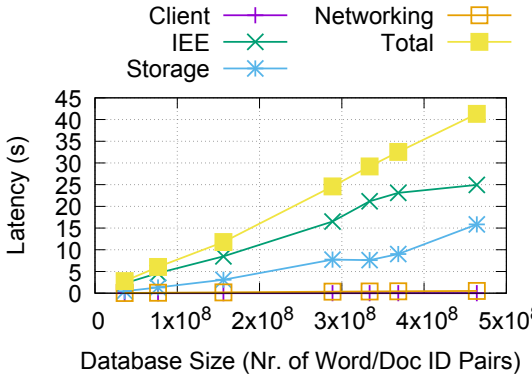

Fig. 4: Performance of each participant in the Search protocol, for an example conjunctive query of five keywords.

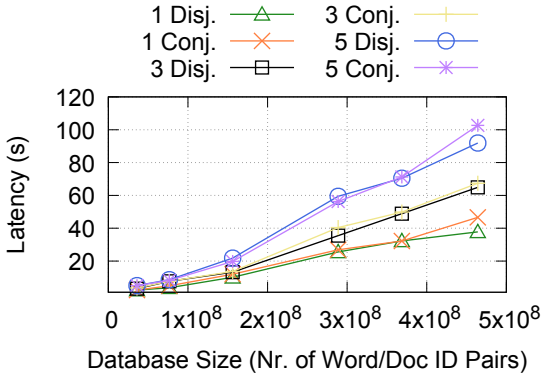

Fig. 5: Impact of the boolean formula and query size on the performance of the Search protocol.

Performance of each Participant. We start by analysing the performance of networking and of each protocol participant in separate when executing the Search protocol. For this analysis we used an example conjunctive query with the five most popular keywords in the database, measuring performance at increasing database sizes. Figure 4 presents the results. In contrast with the previous results for Update, client processing in Search is very efficient. This performance cost is mostly dependent on the query size, nonetheless even for a query of five keywords it is almost close to zero (an average of $80\mu s$). Networking also exhibits similar results.

The remaining performance cost is divided between the storage service and the IEE, with the IEE being the least efficient of the three components. This is due to most computations in Search being performed by the IEE. This aspect can potentially be improved by exploring parallelism in our prototype implementation based on SGX.

Boolean Formulas and Query Size. With this test (Figure 5) we wanted to assess the impact of both the type of operators and the length of the query on overall latency. We used queries in both Conjunctive (CNF) and Disjunctive (DNF) Normal

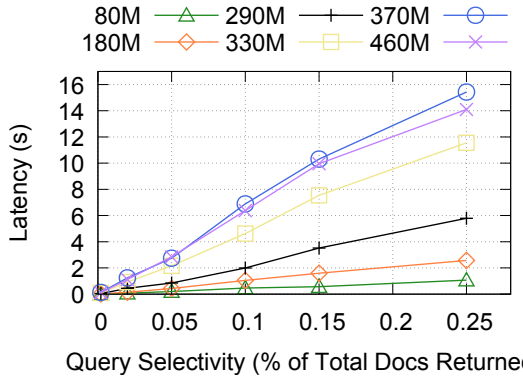

Fig. 6: Impact of query selectivity on the performance of the Search protocol.

Forms, with one, three and five conjunctions and disjunctions. These correspond, for example, to queries of the form $(A \vee B) \wedge (C \vee D)$ (one conjunction) or $(A \vee B) \wedge (C \vee D) \wedge (E \vee F) \wedge (G \vee H)$ (three conjunctions) for CNF; the same logic applies to the DNF.

Analysing the results, we can conclude that BISEN supports queries in any boolean formula with equal performance. For this experiment, the determining factors in performance were the database and query sizes. Increasing the database size leads to a linear increase in the time required for resolving queries, as was already noted in the previous experiment. Moreover, increasing the query size (from one to three and five conjunctions/disjunctions) also increases search latency, but by a smaller fraction. This means performance costs tend to amortize when increasing query sizes.

Query Selectivity. Next we study the impact of query selectivity (i.e., the size of search results) on Search performance. In these experiments, we performed single-keyword queries with different selectivity levels, by choosing query keywords based on their database popularity. Figure 6 shows the results for queries returning from 0.2% to 25% of the database. As expected, query selectivity has a high impact on Search performance. Just by searching a different, more popular keyword, Search performance can go from 1 to 16 seconds. This is not surprising, as more popular keywords appear in more documents, and hence the IEE will have to request, decrypt, and verify additional index entries. Nonetheless, results seem to amortize towards larger databases. These results are also consistent with the performance measurements of Figure 5, whose keyword searches have very high selectivity.

Negations. In Table I we present the impact of negations for queries of fixed size (10 keywords), varying the number of negated keywords – one, five and ten; then a fully negated query – of the form $\neg(A \wedge B)$, and finally the equivalent version of the latter using De Morgan's laws. Our objective was to assess the impact performance of the negation operation across different types of queries and numbers of negations. Results show that the number of negation operations performed has minimal impact, even for larger database sizes, which can be explained by the low overhead of Boolean processing. Since all queries require the same number of entries to be fetched from Storage Service, which is where the main bottleneck lies, their latency is therefore similar.

DB Size	1 Neg.	5 Neg.	10 Neg.	Fully Neg.
35 996 207	4.286	4.498	3.052	4.319
76 672 004	9.335	9.241	9.610	7.185
156 143 147	18.653	18.092	21.095	16.589
333 784 724	52.265	58.227	50.850	51.996
464 054 543	86.057	82.289	85.041	86.938

TABLE I: Performance (in seconds) of negations in the Search protocol.

D. Comparison with IEX-2LEV

We now compare the performance of BISEN with the state of the art in Boolean SSE, in particular the recent IEX-2LEV scheme [25]. To this end, we used the author's open-source implementation [18] (with a filtering parameter of 0.2, as reported in their evaluation [25]), and conducted experiments with the Enron database [27], an email archive with 2.6GB of text data used by the authors.

Since IEX-2LEV requires large volatile storage and was originally evaluated on a machine with 60 GB of RAM and a 60-core CPU, we followed a similar test-bench and deployed IEX-2LEV in our AMD Opteron 6272 CPU with 64 cores and 64GB of RAM. For experimental comparison we deployed BISEN on the same machine, executing IEE computations in SGX simulated mode. Table II presents the results obtained for BISEN and IEX-2LEV, considering increasing database sizes (up to 56238 keyword-document pairs, as we were unable to execute IEX-2LEV with higher database sizes), and different operations: Update (performed as Setup in IEX-2LEV), and Search with queries with eight keywords (selected at random from the Enron database) in both CNF and DNF.

Analysing the results we can conclude that BISEN is much more efficient than the state of the art in Boolean SSE. This phenomenon can be observed both for the Update operation, where IEX-2LEV requires eight hours to index a database with 56 238 pairs while BISEN only requires 0.151 seconds; and the Search operation, where IEX-2LEV is more efficient but still requires 216 seconds to search the largest database with a CNF boolean query while BISEN performs the same query in 0.061 seconds. Furthermore, the improvement in storage performance is also evident from these results, since BISEN could process and index large databases with 10 million pairs in a machine with only 8 GB of RAM and IEX-2LEV could only support little more than 56 thousand pairs in a machine with 64 GB. These results can be explained by the difficulty of managing complex multi-map data-structures that IEX-2LEV needs to employ in order to achieve its security guarantees. In BISEN, by leveraging the natural synergy between standard cryptographic primitives and IEEs deployed as remote trust anchors, we are able to improve performance and scalability by a large fraction, while further improving security and minimizing leakage.

VII. Conclusions

In this paper, we have identified and addressed one of the fundamental security issues in Searchable Symmetric Encryption (SSE) schemes, which is the outsourcing of critical

Database Size (Nr of pairs w/id)	Update		Search CNF		Search DNF	
	BISEN	IEX-2LEV	BISEN	IEX-2LEV	BISEN	IEX-2LEV
9 793	0.151	5143	0.004	12	0.004	15
27 446	0.423	15568	0.021	173	0.012	249
56 238	0.862	29274	0.061	216	0.034	427

TABLE II: Performance comparison between BISEN and IEX-2LEV [25]. All times are in seconds. Queries composed of eight keywords.

cryptographic computations to the untrusted server. This was achieved by proposing a new hybrid approach to SSE that combines standard symmetric-key cryptographic primitives with modern attestation-based trusted hardware. In our approach we minimize assumptions and requirements on the employed hardware technology, in particular regarding its trusted storage capacity. Instead, trusted hardware is used as a limited-capacity Isolated Execution Environment abstraction, extending its resources through standard cryptographic primitives over more abundant (local, or even remote) untrusted resources. Based on this hybrid approach, we proposed BISEN, a new dynamic boolean SSE scheme with both forward and backward privacy, minimal leakage, and optimal computation, storage, and communication overheads. BISEN is shown to be provably secure against active adversaries under the standard security model. Experimental results obtained trough real-world datasets and an open-source implementation of BISEN demonstrate its optimal performance and efficiency properties.

ACKNOWLEDGMENTS

This work was supported by FCT/MCTES through project HADES (PTDC/CCI-INF/31698/2017) and NOVA LINCS (UID/CEC/04516/2013), and the EU through project LightKone (grant agreement n^o 732505).

REFERENCES

[1] T. Alves and D. Felton. TrustZone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi. Secure multiparty computation from SGX. In *Financial Cryptography and Data Security - FC'17*, 2017.

[4] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. Foundations of hardware-based attested computation and application to SGX. In *EURO S&P'16*, pages 245–260, 2016.

[5] M. Bellare and P. Rogaway. Introduction to modern cryptography. *Ucsd Cse*, 207:207, 2005.

[6] D. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT'12*. Springer, 2012.

[7] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A Survey of Provably Secure Searchable Encryption. *ACM CSUR*, 47(2):18:1—-18:51, 2015.

[8] R. Bost. Sophos - Forward Secure Searchable Encryption. In *CCS'16*. ACM, 2016.

[9] R. Bost, B. Minaud, and O. Ohrimenko. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *CCS'17*. ACM, 2017.

[10] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, S. Devadas, et al. Mi6: Secure enclaves in a speculative out-of-order processor. *arXiv preprint arXiv:1812.09822*, 2018.

[11] E. Brickell and J. Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. *International Journal of Information Privacy, Security and Integrity 2*, 1(1):3–33, 2011.

[12] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *CCS'15*, pages 668–679. ACM, 2015.

[13] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS'14*, volume 14, 2014.

[14] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO'13*, pages 353–373. Springer, 2013.

[15] ComScore. The 2017 U.S. Mobile App Report. http://tinyurl.com/ya8kkxan, 2017.

[16] V. Costan and S. Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086, 2016.

[17] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS'06*, pages 79–88, 2006.

[18] Encrypted Systems Lab, Brown University. The clusion library. https://github.com/encryptedsystems/Clusion, 2018.

[19] B. Ferreira, B. Portela, T. Oliveira, G. Borges, J. Leitão, and H. Domingos. BISEN: Efficient Boolean Searchable Symmetric Encryption with Verifiability and Minimal Leakage. Cryptology ePrint Archive, Report 2018/588, 2018. https://eprint.iacr.org/2018/588.

[20] B. A. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. Iron: Functional encryption using intel sgx. In *CCS'17*. ACM, 2017.

[21] T. Frieden. VA will pay $20 million to settle lawsuit over stolen laptop's data. CNN. http://tinyurl.com/lg4os9m, 2009.

[22] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi. Hardidx: practical and secure index with sgx. In *IFIP DBSec*, pages 386–408. Springer, 2017.

[23] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Crypto'16*, pages 563–592. Springer, 2016.

[24] G. Greenwald and E. MacAskill. NSA Prism program taps in to user data of Apple, Google and others. The Guardian. http://tinyurl.com/oea3g8t, 2013.

[25] S. Kamara and T. Moataz. Boolean Searchable Symmetric Encryption with Worst-Case Sub-Linear Complexity. In *EUROCRYPT'17*. IACR, 2017.

[26] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. CRC PRESS, 2007.

[27] B. Klimt and Y. Yang. Introducing the Enron Corpus. In *CEAS*, 2004.

[28] D. Lewis. iCloud Data Breach: Hacking And Celebrity Photos. Forbes. https://tinyurl.com/nohznmr, 2014.

[29] libsodium Deveplopment Team. The sodium crypto library (libsodium). https://libsodium.org, 2018.

[30] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[31] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*, volume 1. Cambridge University Press, 2009.

[32] M. Russinovich. Introducing Azure confidential computing. https://tinyurl.com/y3qqwguk, 2017.

[33] S. Sasy, S. Gorbunov, and C. W. Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. In *NDSS'18*, 2018.

[34] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS'14*, 2014.

[35] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Security'18*. Usenix, 2018.

[36] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *CCS'17*, 2017.

[37] I. Wikimedia Foundation. Wikipedia:Database download. https://en.wikipedia.org/wiki/Wikipedia:Database_download, 2018.

[38] Y. Zhang, J. Katz, and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *Security'16*. USENIX Association, 2016.