HardIDX: Practical and Secure Index with SGX

Benny Fuhry[1], Raad Bahmani[2], Ferdinand Brasser[2], Florian Hahn[1],
Florian Kerschbaum[3], and Ahmad-Reza Sadeghi[2]

[1] SAP Research, {benny.fuhry,florian.hahn}@sap.com,
[2] Technische Universität Darmstadt,
{r.bahmani,f.brasser,a.sadeghi}@trust.tu-darmstadt.de
[3] University of Waterloo, florian.kerschbaum@uwaterloo.ca

Abstract. Software-based approaches for search over encrypted data are still either challenged by lack of proper, low-leakage encryption or slow performance. Existing hardware-based approaches do not scale well due to hardware limitations and software designs that are not specifically tailored to the hardware architecture, and are rarely well analyzed for their security (e.g., the impact of side channels). Additionally, existing hardware-based solutions often have a large code footprint in the trusted environment susceptible to software compromises. In this paper we present HardIDX: a hardware-based approach, leveraging Intel's SGX, for search over encrypted data. It implements only the security critical core, i.e., the search functionality, in the trusted environment and resorts to untrusted software for the remainder. HardIDX is deployable as a highly performant encrypted database index: it is logarithmic in the size of the index and searches are performed within a few milliseconds. We formally model and prove the security of our scheme showing that its leakage is equivalent to the best known searchable encryption schemes.

1 Introduction

Outsourcing the storage and processing of sensitive data to untrusted cloud environment is still considered as too risky due to possible data leakage, government intrusion, and legal liability. The cryptographic solutions Secure Multiparty Computation (MPC) and in particular Fully Homomorphic Encryption (FHE) [23] offer high degree of protection by allowing arbitrary computation on encrypted data, but they are impractical for adoption in large distributed systems [24].

Moreover, there are a number of useful applications that only require a small set of operations. A prime example of such operations is the search and retrieval in an encrypted databases without the need to download all data from the cloud. For this task, different cryptographic schemes have been proposed such as property-preserving encryption [6, 8], or functional encryption [10] and its special case searchable encryption [16, 29, 41]. In this context, performing efficient and secure *range* queries are commonly considered to be very challenging. CryptDB [36] resorts to order-preserving encryption for this purpose which is susceptible to simple ciphertext-only attacks as shown by Naveed et al. [34].

Many schemes for search over encrypted data supporting range queries require search time linear in the number of database records. Recently, schemes with polylogarithmic search time, based on an index structure, have been proposed [17, 19, 29].

In Sect. 7 and Table 2, we elaborate on the search time, query size, storage size and leakage problems of those approaches. Designing an *efficient* searchable encryption scheme with *minimal leakage on the queried ranges* remains an open challenge.

Another line of research [4, 5] leverages the developments in hardware-assisted Trusted Execution Environments (TEEs) for search over encrypted data. Although Intel's recently introduced Software Guard Extension (SGX) [2, 15, 26, 31] has inspired new interest in TEEs, related technologies have been available before, e.g., in ARM processors known as ARM TrustZone [3] as well as in academic research [12, 42]. Also, AMD has recently announced a TEE for their CPUs [27] rising the hope that TEEs will be widely available in x86 processors, and thus in many relevant environments such as clouds, in the near future. TEEs have to interact with untrustworthy components within the same computer system for various reasons. In order to achieve comprehensive security, information leakage through those channels has to be considered and taken care of. Previous SGX based solutions that allow search on encrypted data load and execute the entire unmodified database management system (DBMS) into an enclave [4,5]. They do not formally consider information leakage and do not scale well due to limited memory size of SGX's enclaves and the large footprint of the code they require in the TEE.

Our goal and contribution. We present an efficient scheme for search over encrypted data that can be deployed as a database index. SGX's protection characteristics are utilized to achieve an outstanding tradeoff between security, performance and functionality. The currently fastest software-based schemes that support range queries are [17] and [19]. Our solution significantly improves over these approaches in terms of performance and storage. Compared to the latest hardware-based schemes [4, 5], we improve in terms of security and scalability. Our scheme organizes data in a B^+-tree structure that is frequently used for databases indexes in most database management systems (DMBSs) [37]. Our solution supports searches for single values and value ranges and it can easily be adapted to many other database (search) operations.

We implemented and extensively evaluated our two constructions on SGX-enabled hardware (see Sect. 6). Both have a very small code and memory footprint in the TEE compared to other hardware-based approaches [4,5]. Additionally, our solution scales to arbitrary index sizes as memory usage in the enclave is constant and untrusted resources are used to store the database itself. Our main contributions are as follows:

- Our scheme has logarithmic complexity in the size of index and searches are performed within a few milliseconds.
- We formally model and prove our scheme secure showing that its security (leakage) is comparable to the best known searchable encryption schemes.
- We provide an implementation and evaluate the performance and functional bottleneck of SGX on the basis of two different constructions that are designed specifically for SGX to reduce the Trusted Computing Base.

2 Background

2.1 Intel Software Guard Extensions (SGX)

SGX is an extension of the x86 instruction set architecture (ISA) introduced with the 6th Generation Intel Core processors (code name Skylake). We now present a high level overview of SGX's features utilized by HardIDX (see [2, 15, 26, 31] for more details).

Memory Isolation. On SGX enabled platforms, programs can be divided into two parts, an *untrusted part* and an isolated, *trusted part*. The trusted part, called *enclave* in SGX terminology, is located in a dedicated portion of the physical RAM. The SGX hardware enforces additional protection on this part of the memory. In particular, all other software on the system, including privileged software like OS, hypervisor and firmware cannot access the enclave memory. The (untrusted) host process can invoke the enclave only through a well-defined interface. Furthermore, all isolated code and data is encrypted while residing outside of the CPU package. Decryption and integrity checks are performed when the data is loaded inside the CPU.

Memory Management. SGX dedicates a fixed amount of the system's main memory (RAM) for enclaves and related metadata. For current systems this memory is limited to $128\,\mathrm{MB}$ which is used for both, SGX metadata and the memory for the enclaves themselves. The enclaves can only be deployed in about $96\,\mathrm{MB}$. The SGX memory is reserved in the early boot phase and is static throughout the runtime of the system. The OS can allocate (parts of) the memory to individual enclaves and change these allocation during the runtime of the enclaves. In particular, the OS can swap out enclave pages. SGX ensures integrity, confidentiality and freshness of swapped-out pages.

Attestation. SGX has a remote attestation feature which allows to verify the correct creation of an enclave on a remote system. During enclave creation the initial code and data loaded into the enclave are measured. This measurement can be provided to an external party to prove the correct creation of an enclave. The authenticity of the measurement as well as the fact that the measurement originates from a benign enclave is ensured by a signature, provided by SGX's attestation feature (refer to [2] for details). Furthermore, the remote attestation feature allows for establishing a secure channel between an external party and an enclave.

2.2 Side Channel Attacks

Side channel attacks allow an adversary to extract sensitive information without having direct access to the information source by observing effects of the processing of the sensitive information. They have been known for a long time and various variants have been studied in the past, e.g., hardware side channels, software timing side channels and cache timing side channels [13, 22, 45]. All these attacks are noisy and require repeated execution and measurements to extract the sensitive information.

In the context of SGX, there exist a new class of side channels, called deterministic side channel [44]. As the OS is untrusted, yet still manages the enclave's resources, it can observe the enclaves behavior. In particular, the OS can generate a precise trace of the enclave's code and data accesses at the granularity of pages. In [44] it is shown that this allows to extract sensitive information from an SGX enclave.

3 High Level Design

3.1 HardIDX Overview

The high level design of our solution is shown in Fig. 1. The design involves three entities: the client (who is the data owner and therefore trusted), the untrusted SGX enabled server and the trusted SGX enclave *within* the server.

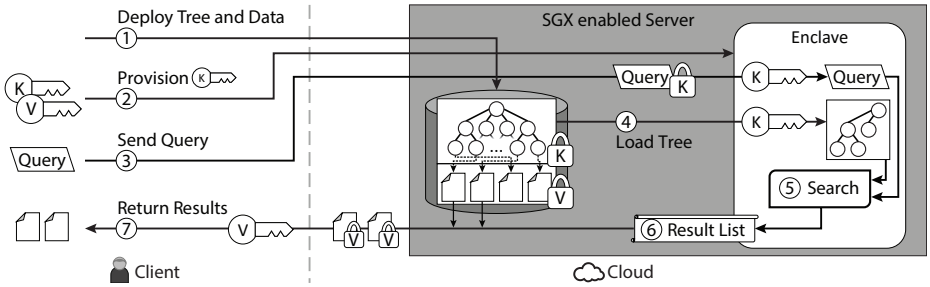


Fig. 1: High level design

Initially, a client prepares its data values by augmenting it with (index) search keys. We abbreviate data values as *values* and the search keys as *keys* throughout this paper. All other values and keys (e.g., cryptographic keys) are clearly differentiate if ambiguous. The values are stored at pseudo-random position. The keys are then inserted into a B^+-tree and the storage order of all nodes is also pseudo-random. The tree and values are linked by adding pointers to the leaves of the tree identifying the random position of the corresponding values. A value can be any data such as records in a relational database or files/documents in other database types. The client then encrypts all nodes of the tree with a secret key SK_k and all values with SK_v. The encrypted B^+-tree and encrypted values are deployed on the untrusted server in the cloud (see step ① in Fig. 1[4]).

The client uses the SGX attestation feature for authenticating the enclave and establishing a secure connection between the client and the enclave (see details in Sect. 2.1). Through this connection, the client provisions SK_k into the enclave (see step ②). This completes the setup of our scheme, which needs to be executed only once.

Now, the client can send (index) search queries to the server that are encrypted with a probabilistic encryption scheme under SK_k. Hence, the untrusted server cannot learn anything about the query, not even if the same query was send before. When a query arrives in the enclave, SK_k is used to decrypt the query (see step ③).

In step ④, the enclave loads the B^+-tree structure (tree nodes, but no values) from the untrusted storage into enclave memory and decrypts it. Given sufficient memory, the entire tree is loaded into the enclave and the search is performed afterwards (see step ⑤). As the tree size can exceed the memory available inside the enclave we provide a second design. In this case, only a subset of tree nodes is loaded into the enclave. The tree is traversed starting from the root node and nodes are fetched from the untrusted storage if necessary. In both cases the search algorithm eventually reaches a set of leaf nodes, which holds pointers to values matching the query. This list of pointers, representing the search result, is passed to the untrusted part (see step ⑥). The untrusted part learns nothing, except for the cardinality of the result set, from this interaction, because the values are stored in a randomized order.

The result of the index search could be processed further, e.g. in combination with additional SQL operators, in the SGX enclave at the server. In order to complete the end-to-end secure search, we assume that the server uses the pointers to fetch the encrypted

[4] For visualization purposes, the tree nodes and values are shown to be encrypted as a block. In reality each node and value is encrypted individually.

values from untrusted storage and sends them to the client, where they are decrypted with SK_v (see step ⑦).

Notably, the plaintext values are never available on the server. They are encrypted with strong standard cryptography methods (AES-128 in GCM mode in our case) and never decrypted on the server, not even inside the SGX enclave. SK_v is only known to the client.

3.2 Assumptions and Attacker Model.

Due to SGX's protection, the attacker cannot directly access the enclave. However, side channels exist through which the attacker could potentially extract sensitive information. We assume the attacker has full control over all software on the system running HardIDX. (1) The attacker can observe all interaction of the enclave with resources outside the enclave. In particular, the attacker can observe the access pattern to B^+-tree nodes stored outside the enclave. (2) The attacker can use deterministic page-fault side channel to observe data access inside the enclave at page granularity [44]. Through this side channel, the attacker can observe access patterns on the B^+-tree stored *inside* the enclave. (3) The attacker can use cache side channel to learn about code paths or data access patterns inside the enclave, as SGX does not protect against them [15].

Hardware attacks are out of scope in this paper. Furthermore, we consider denial of service (DoS) attacks on the cloud server and network out of scope. In this version, we only assume a passive attacker due to page constraints. We present mitigation strategies for an active attacker in the long version [20]. We furthermore assume as single user and the multi user case in Appendix C.

4 Notation and Definitions

4.1 B^+-tree

A B^+-tree is a balanced, n-ary search tree. So called search keys are utilized to index values. A B^+-tree can be used to search for single values, e.g., unique staff ids are used to find the corresponding database record (see Fig. 2) or for ranges, e.g., a salary index that allows to search for all employees falling in a specific salary range.

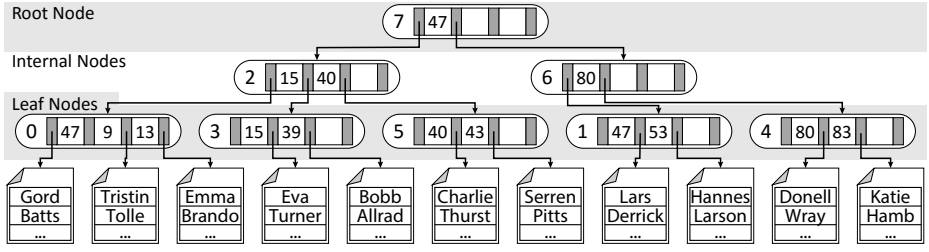


Fig. 2: B^+-tree example: the unique staff ids are used as keys and the values are the staff records (random storage position on the left).

Three node types are differentiated in a B^+-tree: the root node, internal nodes and leaf nodes. Every node x contains $x.\#\mathbf{k}$ keys that are stored in a nondecreasing order:

$x.k_1 \leq ... \leq x.k_{(x.\#\mathbf{k})}$. At every inner node x (including the root if not the only node), the keys separate the key domain into $(x.\#\mathbf{k}+1)$ subtrees that are reachable by $(x.\#\mathbf{k}+1)$ child pointers: $\{x.p_0, ..., x.p_{(x.\#\mathbf{k})}\} = x.\boldsymbol{p}$. Every key $x.k_i$ has a corresponding pointer $x.p_i$ that points to a node containing elements greater than or equal to $x.k_i$ and smaller than any other tag $x.k_j \; \forall j \in [i+1, x.\#\mathbf{k}]$. $x.p_0$ points to a node containing only keys, which are smaller than $x.k_1$. No internal node is linked to a value. Instead, every leaf node x stores $x.\#\mathbf{k}$ keys and a pointer to its corresponding value ($x.p_0$ is not used at the leaves). Every node x in the tree has a unique id $x.id$ and a flag $x.isLeaf$ that stores if the x is a leaf. We denote the B^+-tree without the values as B^+*-tree structure* and p_{x_i} as the storage position of x_i, i.e., the physical memory address.

We use unchained B^+-trees, i.e., the leafs are not connected. Linked leaves would increase the search performance, but it would severely deteriorate the security. The reason is that a range query would directly leak the relationship among leaves if links are followed during a query.

With HardIDX, it is not necessary to define the key domain D in advance as in many other approaches. D can be an arbitrary domain with a defined order relation and a defined minimal and a maximal element recognizable by the algorithms. These two elements, denoted as $-\infty$ and ∞, fulfill the following: $-\infty < x.k < \infty \; \forall x.k \in D$.

The branching factor b specifies a B^+-tree by defining the maximal number of pointers. b also defines the minimal number of pointer for the different node types, but we do not further elaborate on details. For ease of exposition, we assume that every key and pointer fits in an $32\,\mathrm{bit}$ block, but this is no prerequisite for our constructions.

4.2 Probabilistic Symmetric Encryption:

A probabilistic authenticated symmetric encryption consists of three probabilistic polynomial-time algorithms $\texttt{PASE} = \big(\texttt{PASE\_Gen}(1^\lambda), \texttt{PASE\_Enc}(SK, v), \texttt{PASE\_Dec}(SK, C)\big)$ with the usual definitions of functionality. PASE has to be an authenticated IND-CCA secure encryption, e.g., AES-128 in GCM mode.

4.3 Hardware Secured B^+-tree (HSBT)

Based on the presented definition of a B^+-tree, we define the notion of a Hardware Secured B^+-tree (HSBT) as follows. We assume that the B^+-tree should store a set $\mathbf{s}$ of n key-value pairs: $\mathbf{s} = ((k_1, v_1), ..., (k_n, v_n))$. This set consists of n values $\mathbf{v} = (v_1, ..., v_n)$ and their corresponding keys $\mathbf{k} = (k_1, ...k_n)$.

Definition 1 (HSBT)**.** *A secure hardware B^+-tree scheme is a tuple of six polynomial-time algorithms* $\big($HSBT_Setup, HSBT_Enc, HSBT_Tok, HSBT_Dec, HSBT_SearchRange, HSBT_SearchRange_Trusted$\big)$.

Algorithms executed at the client:

$SK \leftarrow \texttt{HSBT\_Setup}(1^\lambda)$**:** *Take the security parameter λ as input and output a secret key SK.*

$\gamma \leftarrow \texttt{HSBT\_Enc}(SK, \mathbf{s})$**:** *Take the secret key SK and a set $\mathbf{s}$ of key-value pairs as input. Output an encrypted B^+-tree γ.*

$\tau \leftarrow \texttt{HSBT\_Tok}(SK, R)$**:** *Take the secret key SK and a range $R = [R_s, R_e]$ as input. Output a search token τ.*

$\mathbf{v}' \leftarrow \texttt{HSBT\_Dec}(SK, \boldsymbol{C'})$**:** *Take the secret key SK and a set of ciphertext $\boldsymbol{C'}$ as input. Decrypt the ciphertexts and output plaintext values $\mathbf{v}'$.*

Executed at the server on untrusted hardware:

$\boldsymbol{C'} \leftarrow$ `HSBT_SearchRange`(τ, γ)**:** *Take a search token τ and an encrypted tree γ as input and call the secure hardware function* `HSBT_SearchRange_Trusted`*. Output a set of encrypted values $\boldsymbol{C'}$.*

Executed at the server on secure hardware:

$\mathbf{P} \leftarrow$ `HSBT_SearchRange_Trusted`$(\tau, \mathbf{X})$**:** *Take a search token τ as input. Output a set of pointers $\mathbf{P}$.*

Definition 2 (Correctness)**.** *Let $\mathcal{D}$ denote a* `HSBT`*-scheme consisting of the six algorithms described in Def. 1. We say that $\mathcal{D}$ is correct if for all $\lambda \in \mathbb{N}$, for all SK output by* `HSBT_Setup`(1^λ)*, for all key-value pairs $\mathbf{s}$ used by* `HSBT_Enc`$(SK, \mathbf{s})$ *to output γ, for all R used by* `HSBT_Tok`(SK, R) *to output τ, for all $\boldsymbol{C'}$ output by* `HSBT_SearchRange`(τ, γ)*, the values $\mathbf{v}'$ output by* `HSBT_Dec`$(SK, \boldsymbol{C'})$ *are all values in $\mathbf{s}$ for which the corresponding keys $\mathbf{k}'$ fall in R, i.e., $\mathbf{v}' = \{v_i | (k_i, v_i) \in \mathbf{s} \wedge k_i \in [R_s, R_e] = R\}$.*

Our security model, which we define next, is based on a proof framework introduced by Curtmola et al. in [16]. A description about the proof technique can be found in Appendix A. At security models of searchable encryption schemes so far, the leakage only covers the transaction between the client and server. In our scenario, there is an additional transaction between the server and the secure hardware that can be viewed by the adversary. Therefore, we extend the CKA2-security to CKA2-HW-security by introducing a new type of leakage denoted as $\mathcal{L}_{hw}$. It consists of the inherent leakage of the used secure hardware and the inputs/outputs to/from the secure hardware.

Definition 3 (CKA2-HW-security)**.** *Let $\mathcal{D}$ denote a* `HSBT`*-scheme consisting of the six algorithms described in Def. 1. Consider the probabilistic experiments $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$, whereas $\mathcal{A}$ is a stateful adversary and $\mathcal{S}$ is a stateful simulator that gets the leakage functions $\mathcal{L}_{enc}$ and $\mathcal{L}_{hw}$.*

$\mathbf{Real}_{\mathcal{A}}(\lambda)$**:** *the challenger runs* `HSBT_Setup`(1^λ) *to generate a secret key SK. $\mathcal{A}$ outputs a set of key-value pairs $\mathbf{s}$. The challenger calculates $\gamma \leftarrow$* `HSBT_Enc`$(SK, \mathbf{s})$ *and passes γ to $\mathcal{A}$. Afterwards, $\mathcal{A}$ makes a polynomial number of adaptive queries for arbitrary ranges R. The challenger returns search tokens τ to $\mathcal{A}$ after calculating $\tau \leftarrow$* `HSBT_Tok`(SK, R)*. $\mathcal{A}$ can use γ and the returned tokens at any time to make queries to the secure hardware. The secure hardware returns a set of pointers $\mathbf{P}$. Finally, $\mathcal{A}$ returns a bit b that is the output of the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$**:** *the adversary $\mathcal{A}$ outputs a set of key-value pairs $\mathbf{s}$. Using $\mathcal{L}_{enc}$, $\mathcal{S}$ creates γ and passes it to $\mathcal{A}$. Afterwards, $\mathcal{A}$ makes a polynomial number of adaptive queries for arbitrary ranges R. The simulator $\mathcal{S}$ creates tokens τ and passes them to $\mathcal{A}$. The adversary $\mathcal{A}$ can use γ and the returned tokens at any time to make queries to $\mathcal{S}$ (that simulates the secure hardware). $\mathcal{S}$ is given $\mathcal{L}_{hw}$ and returns a set of pointers $\mathbf{P}$. Finally, $\mathcal{A}$ returns a bit b that is the output of the experiment.*

We say $\mathcal{D}$ is $\left(\mathcal{L}_{enc}, \mathcal{L}_{hw}\right)$-secure against adaptive chosen-keyword attacks if for all probabilistic polynomial-time algorithms $\mathcal{A}$, there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that

$$|\Pr\left[\mathbf{Real}_{\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1\right]| \leq \mathrm{negl}(\lambda)$$

5 Search Algorithms

In this section, we will present two different constructions that enable a client to the search for a single value or a range of values based on keys. We use B^+-trees in both constructions to achieve logarithmic search and SGX to protect the confidentiality and integrity of the data.

5.1 Construction 1

We sketch our first correct (according to Def. 2) and secure (according to Def. 3) construction in this section. A more detailed construction can be found in the long version of the paper [20]. The guiding idea of the construction is that the entire data should be stored and processed inside the enclave. The client constructs the B^+-tree locally, encrypts the B^+-tree structure and the values with SK_k and SK_v, respectively (see 5.2 for details). Both are sent to the cloud provider and the SGX application consisting of a trusted and an untrusted part get deployed there.

Software measurement as described in Sect. 2.1 is used for remote attestation, i.e., to prove to the client that the correct software is deployed on an SGX enabled CPU. During the deployment, the application reserves an SGX protected memory region (see Sect. 2.1). A secure transfer protocol between client and server (see Sect. 2.1) is used to deploy SK_k inside the enclave. Thus SK_k is only known by the enclave's process and the client. The cloud provider and all other processes cannot access this key at any point in time. The next step is to load the B^+-tree structure (tree nodes, but no values) from an untrusted to the isolated memory region and use SK_k to decrypt the tree nodes. It is important to note that the tree is still protected, because all data inside the enclave is secured by SGX. The values are stored outside of the enclave to save enclave memory. This has no security implications as the values are encrypted with an authenticated IND-CCA secure encryption scheme and they are stored in a randomized order. The enclave is then ready to receive search query tokens τ from the client.

The untrusted part relays τ to the trusted part. There, the token gets decrypted and a B^+-tree traversal is performed. All pointers that lead to values falling in the queried range are returned in random order. The untrusted part receives pointers to the values, loads the values from memory or disk and passes them to the client.

This construction suffers from the substantial problem mentioned before: the memory reserved for SGX is limited to $128\,\mathrm{MB}$, and only about $96\,\mathrm{MB}$ can be used for data and code. SGX supports a larger enclave size, but enclave pages have to be swapped in and out in this case. Our evaluation (see Sect. 6) shows that even $50{,}000{,}000$ values are possible. A B^+-tree, however, is only a small part of a full encrypted DBMS based on our constructions. Other components would occupy large regions of the restricted enclave memory and thus further limit the available space.

Construction 1 provides a very low leakage (see [20] for details), but it is not usable in a big data cloud scenario, because of the described limitation. Therefore, we present a second construction in the next section.

5.2 Construction 2

In this section, we describe our second correct (according to Def. 2) and secure (according to Def. 3) construction. Instead of loading all nodes into the enclave, the main idea

is to only load the nodes required to traverse the tree. The challenge is to optimize the communication bottleneck between the untrusted part and the enclave. We performed extensive benchmarking and algorithm engineering in order to identify and minimize the most important run-time consuming tasks, such as switching between the untrusted part and the enclave. The decisive advantage of our second construction is that the required memory space inside the enclave is $O(1)$ for a tree of arbitrary size. The trade-off is that all nodes are stored encrypted inside untrusted main memory or on the untrusted disk and thus have to be decrypted by the enclave before processing. This also leads to a slightly larger leakage than in the first construction, namely a finer-granular access pattern on node instead of page level (details are described by a formal model and proof later).

The setup of the HSBT-scheme is slightly different than in the first construction in order to implement the described features. As before, the B^+-tree is constructed and encrypted at the client and is then transferred to the cloud provider. However, the application does not reserves memory for the whole B^+-tree structure inside the enclave. Instead, it only reserves a fixed space, denoted as `reservedSpace`, for on the fly processing. The remote attestation and secure key deployment are performed as in the previous construction.

We now describe an HSBT-scheme consisting of six algorithms $\big($HSBT2_Setup, HSBT2_Enc, HSBT2_Tok, HSBT2_Dec, HSBT2_SearchRange, HSBT2_SearchRange_Trusted$\big)$ that utilizes a pseudorandom permutation $\Pi : \{0,1\}^\lambda \times \{0,1\}^{log_2 \#\mathbf{x}} \rightarrow \{0,1\}^{log_2 \#\mathbf{x}}$ in more detail. Note that all but HSBT2_SearchRange and HSBT2_SearchRange_Trusted exactly match the algorithms utilized for Constr. 1.

$SK \leftarrow$ HSBT2_Setup(1^λ)**:** Use input λ to execute PASE_Gen two times and output $SK = \big(SK_k, SK_v\big)$. SK_v and SK_k are kept secret at the client. SK_k is additionally shared with the server enclave using a secure transport and deployment protocol. The enclave stores SK_k inside the isolated enclave.

$\gamma \leftarrow$ HSBT2_Enc$(SK, \mathbf{s})$**:** Take SK and $\mathbf{s} = ((k_1, v_1), ..., (k_n, v_n))$ as input. Start by storing all values $\mathbf{v} = (v_1, ..., v_n)$ in a random order. An almost standard B^+-tree insertion is used for all keys. One difference is that every newly created node x gets an id according to the creation order, i.e., the first node gets id 0 ($x.id = 0$), the second id 1 ($x.id = 1$) et cetera. After each pair is inserted, the empty position for keys and pointers in the tree get filled up. More specifically, a node x that contains $x.\#\mathbf{k}$ keys from the domain gets filled with $(b - 1 - x.\#\mathbf{k})$ keys ∞ and $(b - x.\#\mathbf{k})$ dummy pointers. Then, all keys and pointers are padded to a length of $32\,\mathrm{bit}$ (this is no prerequisite of our solution). The ids are used by the algorithm to store the nodes at pseudorandom positions: $p_x = \Pi(SK_k, x.id)$. Now, we have a B^+-tree in which every node occupies the same storage space and the order of the nodes and values is random. Finally, PASE_Enc$(SK_v, \cdot)$ is used to encrypt every value and PASE_Enc$(SK_k, \cdot)$ is used to encrypt every node. The encrypted nodes and values form the encrypted tree γ, which is protected by an authenticated IND-CPA secure encryption.

$\tau \leftarrow$ HSBT2_Tok(SK_k, R)**:** Use input SK_k and $R = [R_s, R_e]$ to calculate $\tau \leftarrow$ PASE_Enc$(SK_k, R_s || R_e)$ and output τ. Queries for all elements below R_e or all elements above R_s can be created by using $R_s = -\infty$ or $R_e = \infty$, respectively.

$\mathbf{v}' \leftarrow$ HSBT2_Dec$(SK_v, \boldsymbol{C'})$**:** Use input SK_v to decrypt the encrypted values $\boldsymbol{C'} = \big(C_0, ..., C_j\big)$: $\mathbf{v}' = \big($PASE_Dec$(SK_v, C_0), ...,$ PASE_Dec$(SK_v, C_j)\big)$. Output $\mathbf{v}'$.

$\boldsymbol{C'} \leftarrow$ `HSBT2_SearchRange`(τ, γ)**:** Take the search token τ and the encrypted tree γ as input. At the beginning, pass only the root node to the trusted part and receive pointers to nodes that should be traversed next. The trivial solution is to pass one node after another. A problem with this design is that every context switch from the untrusted to the trusted part or back causes an overhead. We therefore optimized the number of context switches: transfer as many nodes as currently in the queue, but not more than fit into `reservedSpace`. We denote the maximal number of nodes as `maxAmount`, which is directly influenced by `reservedSpace`: `maxAmount` $=$ `reservedSpace`$/(o \cdot 128\,\text{bit})$ where o is the number of AES-blocks used by each node. Nodes are passed until no further are requested. Then output $\boldsymbol{C'}$ by dereferencing pointers to the values. See Algo. 1 for details.

$\mathbf{P} \leftarrow$ `HSBT2_SearchRange_Trusted`$(\tau, \mathbf{X})$**:** Take a search token τ and nodes $\mathbf{X}$ as input. During the setup phase, SK_k was deployed inside the secure hardware. Therefore, the algorithm is able to decrypt all nodes and the token that are encrypted with SK_k. Then, search all keys falling in the query range, whereby *all* keys are accessed. Finally, return the corresponding pointers in a random order together with the plaintext $isLeaf$ tag for each pointer. The tag is necessary, because the untrusted part has no direct access to this encrypted node information. See Algo. 2 for details.

Algorithm 1 `HSBT2_SearchRange`(τ, γ)

```
1: X = ∅                                          ▷ FIFO queue
2: X.ENQUEUE(root)
3: results = ∅
4: while X ! = ∅ do
5:     for i=0; i < X.size && i < maxAmount; i++ do
6:         X_tmp = X.DEQUEUE( )
7:     end for
8:     results_tmp = HSBT2_SEARCHRANGE_TRUSTED(τ, X_tmp)
9:     for (isLeaf, p) in results_tmp do
10:        if isLeaf then
11:            results.ADD(*p)
12:        else
13:            X.ENQUEUE(*p)
14:        end if
15:    end for
16: end while
17: return results
```

Algorithm 2 `HSBT2_SearchRange_Trusted`$(\tau, \mathbf{X})$

```
1: τ_Plain = PASE_Dec(SK_k, τ)
2: parse τ_Plain as (R_s, R_e)
3: X_tmp = {PASE_Dec(SK_k, *X_0), PASE_Dec(SK_k, *X_1), ...}
4: P = ∅
5: for x in X_tmp do
6:     if not x.isLeaf and R_s < x.k_1 then
7:         P.ADD(x.p_0)
8:     end if
9:     for i = 1, i < b − 1, i++ do
10:        if (x.k_i ≤ R_s < x.k_{i+1}) ||
           (x.k_i ≤ R_e < x.k_{i+1}) ||
           (R_s ≤ x.k_i && x.k_{i+1} ≤ R_e) then
11:            P.ADD(x.p_i)
12:        end if
13:    end for
14:    if R_e ≥ x.k_{b−1} then
15:        P.ADD(x.p_{b−1})
16:    end if
17: end for
18: P = random permutation of P
19: return (*P_0.isLeaf, P_0), (*P_1.isLeaf, P_1), ...
```

The construction is correct according to Def. 2, because it is based on a textbook B^+-tree traversal. The difference to the textbook algorithm is that the nodes are loaded inside the enclave after another and that each node is encrypted. These changes do not influence the correctness, because each node remains accessible to the enclave and the encryption (at the client) and the decryption (inside the enclave) are based on a correct PASE-scheme.

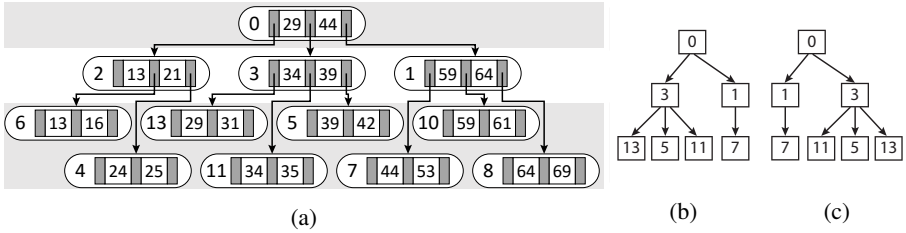
Next, we will prove the security of Constr. 2. The first step is to define the leakage functions that are based on the attack model described in Sect. 3.

$\mathcal{L}_{enc}(\mathbf{s})$**:** Given the key-value pairs $\mathbf{s} = ((k_1, v_1), ...(k_n, v_n))$, this function outputs the amount n of values, the size of each value and the amount of B^+-tree nodes $\#\mathbf{x}$.

$\mathcal{L}_{hw}(\mathbf{s}, T, R, t)$**:** Given the key-value pairs $\mathbf{s}$, the plaintext B^+-tree T, the search range R and given point in time t, this function outputs the nodes access pattern $\mathcal{X}(\mathbf{s}, T, R, t)$ and the value pointers access pattern $\Delta(\mathbf{s}, T, R, t)$.

The nodes access pattern $\mathcal{X}(\mathbf{s}, T, R, t)$ is a tree that contains the storage positions of all nodes in T that get accessed when searching for the range R. For a more formal definition, we denote the set of leaf nodes that contain keys from the range as M, i.e., $M = \{x \,|\, x \in T \,\wedge\, x.isLeaf \,\wedge\, x.k_j \in R,\, j \in [1, b-1]\}$. Additionally, we define $x{\rightarrow}parent_1$ as the parent node of x and $x{\rightarrow}parent_j$ denotes the node that is reached by moving j layers up in the tree starting from x. We denote a node that only contains the storage position of a node x_i as y_i. Now, we can specify the node set $\mathbf{Y}$ of $\mathcal{X}$ as $\mathbf{Y} = \{y_i \,|\, x_i \in M\} \cup \{y_i \,|\, x_i \in T \wedge x \in M \wedge x_i == x{\rightarrow}parent_j, j \in [1, h-1]\}$. The set of directed edges in $\mathcal{X}$ is $\{(y_i, y_j) \,|\, y_i, y_j \in \mathbf{Y} \wedge \exists\, x_i, x_j \in T : x_i == x_j{\rightarrow}parent_1\}$. The time parameter t defines a snapshot of the random (but fixed) order of sibling nodes at a given point in time. See Fig. 3 for an illustrative example.

The value pointers access pattern $\Delta(\mathbf{s}, T, R, t)$ is defined as the pointers to the result values together with the leaf nodes in which these pointers are stored. More formally, $\Delta(\mathbf{s}, T, R, t) = \{(x, \mathbf{P}_x) \,|\, x \in T \wedge x.isLeaf \wedge \exists\, x.k_j \in R, j \in [1, b-1] \wedge \mathbf{P}_x = \{x.p_l \,|\, x.k_l \in R, l \in [1, b-1]\}\}$. The time parameter t defines a random but fixed order of the pointers.



(a) (b) (c)

Fig. 3: Illustration of nodes access pattern leakage: (a) example B^+-tree T (storage positions on the left), (b) leakage $\mathcal{X}(\mathbf{s}, T, R, t)$ for $R = [33, 55]$ and B^+-tree T at t_1, (c) leakage $\mathcal{X}(\mathbf{s}, T, R, t)$ for $R = [33, 55]$ and B^+-tree T at t_2

Theorem 1 (Security)**.** *The secure hardware B^+-tree construction presented above is $(\mathcal{L}_{enc}, \mathcal{L}_{hw})$-secure according to Def. 3.*

The idea of the security proof is to describe how a polynomial-time simulator $\mathcal{S}$ simulates the encrypted B^+-tree γ, the token τ and the secure hardware so that a PPT adversary $\mathcal{A}$ can distinguish between $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with at most negligible probability. The detailed proof can be found in Appendix B.

The leakage definition and security proofs for Constr. 1 are similar. The main difference is the granularity of the tree and value pointers access pattern. In Constr. 2, the attacker is able to reveal accesses on a node level. In contrast, the attacker in Constr. 1 is able to reveal accesses on page level, because SGX inherently leaks the page access pattern.

Note that the page access leakage is an upper bound in the first construction. Each page allocates $4\,\mathrm{kB}$ and every encrypted node consists of one or multiple AES-blocks. Up to $k = 4\,\mathrm{kB}/(o \cdot 128\,\mathrm{bit})$ nodes are contained in one page if o AES-blocks are used by each node. Experiments showed that 102 AES-blocks are used for each node if $b = 100$ and $32\,\mathrm{bit}$ keys and pointers are used. Therefore, even multiple of those huge nodes fit within a single page.

5.3 Side Channels

Our implementation is concerned with three types of (side) channels: external resource access, page-fault side channel and cache timing side channel (see Sect. 3). By means of all three channels an adversary can observe access patterns to memory with the goal of inferring sensitive information from the observed access patterns.

By monitoring access to external resources, the attacker tries to gain information about the tree structure and ultimately the order of values stored in the database. The only external resources accessed by Constr. 1 are the encrypted values stored at random positions, which does not leak information. Constr. 2 also accesses B^+-tree nodes but this is explicitly covered in its leakage.

The page-fault side channel allows the attacker to reliably observe memory access patterns at a granularity of $4\,\mathrm{kB}$. All accesses within the same page are indistinguishable for the attacker and, thus, are not exploitable. The implementation of Constr. 1 explicitly considers the leakage of the tree structure through this side channel. In Constr. 2, this side channel does not leak additional information, as nodes are smaller than memory pages and the nodes access pattern is leaked anyway by storing the B^+-tree outside of the enclave.

Cache timing side channel allow finer grained memory access observations while being less reliable. Nevertheless, assuming an adversary who is able to observe accesses within a node, the attacker needs to determine which links to child nodes are followed. Our algorithm, however, accesses every key and pointer, whether the pointer is followed or not. By this and other fine grained implementation details, we achieve data independent accesses and thwart the cache timing side channel.

Leakage of cryptographic keys are thwarted for page-fault and cache timing side channel by using leakage resilient implementations and hardware features [7]. For instance, the AES implementation used in HardIDX holds the S-Boxes in CPU registers instead of RAM to hamper cache side channel attacks [32].

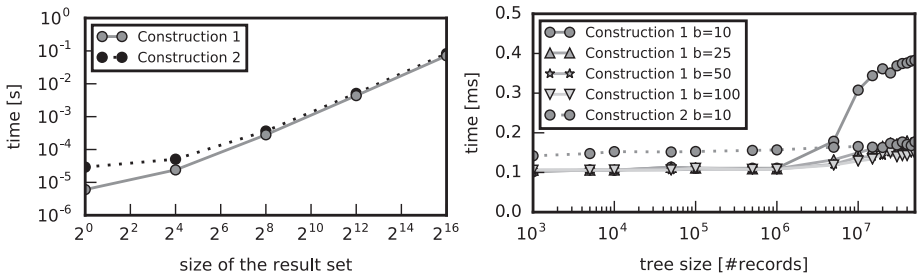
6 Performance Evaluation

In this section, we present our evaluation results collected in a number of experiments with real SGX hardware. Our evaluation system was equipped with an Intel Core i7-6700 processor at $3.40\ \mathrm{Ghz}$ and $32\ \mathrm{GB}$ DDR4 RAM. 64-bit Ubuntu 14.04.1 extended with SGX support was used as operating system.

6.1 Construction 1 vs. Construction 2

First, we compare the performance of our two constructions. The parameters of the B^+-tree are held constant for this comparison: the branching factor is 10 and the tree contains $1{,}000{,}000$ key-value pairs. Queries with five different sizes of the result set

are used: $2^0, 2^4, 2^8, 2^{12}, 2^{16}$. The search ranges were selected uniformly at random and every result size is tested with $1{,}000$ different ranges. Figure 4a depicts the results of this evaluation, whereby the x-axis shows the size of the result set and the y-axis shows the median of the run-times in ms.



(a) (b)

Fig. 4: (a) Comparison of constructions and (b) effect of different branching factors

The performance difference can be explained by the following effects:

- **Processor mode switch.** Before executing inside an enclave, the processor has to switch into "enclave mode". This includes, e.g., storing the current CPU context on the host process' stack and loading the CPU context of the enclave. In Constr. 1 only one switch is required, whereas in the Constr. 2 $O(\log_b n)$ switches are performed, as at least each level of the B^+-tree is loaded into the enclave.
- **Data transfer.** In Constr. 1, the data transfer between trusted and the untrusted code is limited to the result set and the query whereas in Constr. 2 also part of the B^+-tree is transferred between the two components.
- **Access to plain data.** In Constr. 1, decryption is a one-time effort after loading the entire B^+-tree into the enclave. During query processing, it has access to plaintext nodes of the B^+-tree. Constr. 2 incrementally loads the B^+-tree nodes from untrusted storage. All processed nodes need to be decrypted during query processing.

Construction 2, therefore, is slower than Constr. 1 by a small factor at any result size. For an increasing size of the result set, both algorithms search a linearly increasing part of the tree. Figure 4a shows that the run-times of our two constructions converge (on a logarithmic scale). This shows that the effects described above diminish compared to the search time of the algorithm.

6.2 Memory Management

In order to identify the limiting parameters in the memory management of our two constructions, we evaluate B^+-trees with different tree sizes (amounts of key-value pairs) and branching factors. On each tree we ran $1{,}000$ randomly chosen queries with result set size of 100 and tested with the branching factors 10, 25, 50 and 100. The results of these evaluation are depicted in Fig. 4b. The x-axis shows the size of the B^+-tree and the y-axis shows the median run-time of the queries.

We see a sharp increase of the run-time above a tree size of 10^6 records. This is due to the exhausted memory in SGX and the virtual memory mechanism of the operating system that swaps pages in and out. This is not security critical, since pages remain

encrypted and integrity protected by the SGX system, even when they are swapped out of the SGX protected memory. The figure also reveals a significant difference in the impact of paging between different branching factors. The reason becomes clear by considering the number of required page swaps. The lower the branching factor, the higher the number of nodes in a B^+-tree. The higher the number of nodes, the higher the number of accesses to different memory pages. The higher the number of different page accesses, the higher the probability of a swapped out page.

We also see that Constr. 2 is not affected by paging, albeit supporting an unlimited tree size. Our data also shows that, as expected, higher branching factors result in better performance and the runtime approaches the runtime of Constr. 1.

6.3 Comparison with Related Work

In this section, we compare our Constr. 2 against the currently fastest approach with comparable security features and a security proof presented by Demertzis et al. in [17]. The authors present seven different constructions that support range queries. The constructions have different tradeoffs regarding security, query size, search time, storage and false positives. We do not compare against the highly secure scheme with prohibitive storage cost and also exclude the approaches with false positives as our construction does not lead to false positives. Instead, we compare against the most secure approach without these problems: Logarithmic-URC.

We assume that the OXT construction from [14] is used as underlying symmetric searchable encryption scheme (SEE) by Logarithmic-URC. Fundamentally, the SSE scheme is changeable, but the authors of [17] also utilize OXT for the security and performance evaluation. One has to note that a quite equal construction as Logarithmic-URC was presented independently by Faber et al. in [19]. We implemented the algorithm of [17], but a security and performance comparison to [19] would lead to comparable results.

Table 1 compares our Constr. 2 and Logarithmic-URC. In this evaluation, we use a branching factor of 100 for Constr. 2 and search for a randomly chosen range that contains 100 results. Every test for the four different tree sizes (100, 1,000, 10,000, 100,000) was performed 1,000 times and the table shows the mean.

Table 1: Time comparison of random range queries with Logarithmic-URC [17] and our Constr. 2

Tree Size	100	1,000	10,000	100,000
Logarithmic-URC	0.015 s	0.020 s	0.051 s	1.052 s
Const. 2 ($b = 100$)	0.119 ms	0.121 ms	0.124 ms	0.125 ms

Our construction runs in about a tenth of a millisecond and with very moderate increase for all tree sizes. In contrast, Logarithmic-URC requires at least multiple milliseconds up to a seconds for bigger trees. A reason for the performance difference might be that OXT construction itself is less efficient then our construction. Furthermore, the search time of OXT depends on the number of entries. Logarithmic-URC fills the OXT construction with elements from a binary tree over the domain for every stored key. An increasing domain severely increases the tree height of a binary tree and thus the

number of entries for OXT. In contrast, the height of the B^+-tree in our construction increases much slower with the tree size.

It is not trivial to compare Logarithmic-URC and Constr. 2 regarding security. The access pattern leakage and the leakage of the internal data structure of Logarithmic-URC is comparable to our access pattern leakages. However, Logarithmic-URC additionally leaks the domain size, the search range size and the search pattern. The search pattern reveals whether the same search was performed before, which might be sensitive information.

7 Related Work

7.1 Searchable Encryption

Searchable encryption scheme supporting range queries are rare. Table 2 shows a comparison of different searchable encryption schemes and other schemes that support range queries. Note that all existing range-searchable encryption schemes leak the access pattern – including ours. The first range-searchable scheme by Boneh and Waters in [11] encrypt every entry linear in the size of the plaintext domain. The first scheme with logarithmic storage size per entry in the domain was proposed by Shi et al. in [40]. Their security model is somewhat weaker than standard searchable encryption. The construction is based on inner-product predicate encryption which has been made fully secure by Shen et al. in [39]. All of these schemes have linear search time.

Lu built the range-searchable encryption from [39] into an index in [29]. He enabled polylogarithmic search time, but his encrypted inverted index tree reveals the order of the plaintexts and is hence only as secure as order-preserving encryption. Wang et al. [43] proposed a multi-dimensional extension of Lu [29], but it suffers from the same problem of order leakage. There is no known searchable encryption schemes for ranges – until ours – that has polylogarithmic search time and leaks only the access pattern.

ORAM can in principle be used to hide the access pattern of searchable encryption. However, Naveed shows that the combination of the two is not straightforward [33]. Special ORAM techniques, like TWORAM [21], are needed.

Table 2: Comparison of range-searchable encryption schemes. n is the number of keys, D is the size of the plaintext domain and R is the query range size.

Scheme	Search time	Query Size	Storage Size	Search Pattern Leakage	Order Leakage
Boneh, Waters [11]	$O(nD)$	$O(D)$	$O(nD)$	yes	no
Shi et al. [40]	$O(n \log D)$	$O(\log D)$	$O(n \log D)$	yes	no
Shen et al. [39]	$O(n \log D)$	$O(\log D)$	$O(n \log D)$	no	no
Lu [29]	$O(\log n \log D)$	$O(\log D)$	$O(n \log D)$	no	yes
Demertzis et al. [17] Faber et al. [19]	$O(\log R)$	$O(\log R)$	$O(n \log D)$	yes	no
This papers	$O(\log n)$	$O(1)$	$O(n)$	no	no

7.2 Encrypted Databases

Encrypted databases, such as CryptDB [36], use property-preserving encryption for efficient search. Property-preserving encryption has very low deployment and runtime

overhead due to the ability to use internal index structures of the database engine in the same way as on plain data. Order-preserving encryption [1,8,9,28] allows range queries on the ciphertexts as on the plaintexts. However, Naveed et al. [34] initiated the research direction of practical ciphertext-only attacks on property-preserving encryption, in particular order-preserving encryption, which recover the plaintext in many cases with very high probability (close to 100%) and further attacks followed [18, 25].

Cash et al. [14] introduce a new protocol called OXT that allows evaluation of boolean queries on encrypted data. Faber et al. [19] extend this data structure to support range queries but either leak additional information on the queried range or the result set contains false positives. In [17], Demertzis et al. present several approaches for range queries. We provide an experimental and detailed comparison in Sect. 6.3.

7.3 TEE-Based Applications

Trusted Database System (TDB) uses a trusted execution environment (TEE) to operate the entire database in a hostile environment [30]. While TDB encrypts the entire database storage and metadata, it is not concerned with information leakage from the TEE. Neither does TDB aim at hiding access patterns nor does it consider side channels attacks against the TEE. Furthermore, since the entire DB operates in the TEE the trusted computing base is very large exposing a very large attack surface.

Haven is an approach to shield application on an untrusted system using SGX [5]. The goal of Haven is to enable the execution of unmodified applications inside an SGX enclave. This technique could be used to isolated off-the-shelf databases with SGX, however, Haven does not consider information leakages through memory access patterns or interactions with the untrustworthy outside world. Furthermore, this approach limits the size of the database due to limited enclave size.

VC3 adapts the MapReduce computing paradigm to SGX [38]. There the data flows between Mapper and Reducer can leak sensitive information and it is excluded from their adversary model. In contrast, we focus on information leakage in the interaction of an enclave with other entities.

In [35], SGX protected machine learning algorithms have been adapted to prevent the exploitation of side channels by the usage of data-oblivious primitives. Access to external data is addressed by randomizing the data and always accessing all data, i.e., their solution has an complexity of $O(n)$, even for tree searches.

8 Conclusion

In this paper, we introduce HardIDX – an approach to search for ranges and values over encrypted data using hardware support making it deployable as a secure index in an encrypted database. We provide a formal security proof explicitly including side channels and an implementation on Intel SGX. Our solution compares favorably with existing software- and hardware-based approaches. We require few milliseconds even for complex searches on large data and scale to almost arbitrarily large indices. We only leak the access pattern and our trusted code protected by SGX hardware is very small exposing a small attack surface.

Acknowledgments. This research was co-funded by the German Science Foundation, as part of project P3 within CRC 1119 CROSSING, the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 644412 (TREDISEC) and No. 643964 (SUPERCLOUD), and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

A Proof Framework

In [16], Curtmola et al. introduced a three step framework to proof the security of searchable encryption. The first step is to formulate a leakage, i.e., an upper bound of the information that an adversary can gather from the protocol. Secondly, one defines the $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and a $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ game for an adaptive adversary $\mathcal{A}$ and a polynomial time simulator $\mathcal{S}$. $\mathbf{Real}_{\mathcal{A}}(\lambda)$ is the execution of the actual protocol and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ utilizes $\mathcal{S}$ to simulate the real game by using only the formulated leakage. An adaptive adversary can use information learned in previous protocol iterations for its queries. Third, a scheme is CKA2-secure if one can show that $\mathcal{A}$ can distinguish the output of the games with probability negligibly close to 0. This in turn means that $\mathcal{A}$ does not learn anything besides the leakage stated in the first step, because otherwise he could use this additional information to distinguish the games.

B $\big(\mathcal{L}_{enc}, \mathcal{L}_{hw}\big)$-security Proof

In Sect. 5.2, we provide the description of Constr. 2 and the leakage of the construction. We now prove Theorem 1 by describing a polynomial-time simulator $\mathcal{S}$ for which a PPT adversary $\mathcal{A}$ can distinguish between $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with negligible probability.

Proof. The simulator $\mathcal{S}$ works as follows:

- *Setup:* $\mathcal{S}$ creates a new random key $\widetilde{SK} = \texttt{PASE\_Gen}(1^\lambda)$ and stores it.
- *Simulating γ:* $\mathcal{S}$ gets $\mathcal{L}_{enc}$ and creates $\#\mathbf{x}$ nodes $\mathbf{X} = \big(x_1, ..., x_{\#\mathbf{x}}\big)$ filled with random keys, random pointers and increasing node ids. These nodes are stored in the pages $\big(\omega_1, ...\omega_{\#\omega}\big)$. Additionally, $\mathcal{S}$ generates n encryptions of random values $\boldsymbol{C} = \big(C_1, ..., C_n\big)$ using PASE_Enc, the number of values and the size of the values. Every encrypted value is given a distinct index. $\mathcal{S}$ outputs $\gamma = (\mathbf{X}, \boldsymbol{C})$
  All described operations can be executed by $\mathcal{S}$, because the information required for the encryption of values is included in the leakage. The simulated γ has the same size as the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and the IND-CCA security of PASE makes the nodes and values indistinguishable from the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$.
- *Simulating τ:* The simulator $\mathcal{S}$ creates two random values r_1 and r_2 and encrypts them: $\tau \leftarrow \texttt{PASE\_Enc}(SK_k, R_s||R_e)$. $\mathcal{S}$ outputs τ.
  The simulated τ is indistinguishable from the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$ as a result of the IND-CCA security of PASE.
- *Simulating secure hardware:* At time t, the simulator $\mathcal{S}$ receives encrypted nodes (denoted as $\mathbf{X}$), a token τ and $\mathcal{L}_{hw}$. It has to simulate the output of the secure hardware enclave. The simulator decrypts every node $x_i \in \mathbf{X}$ with $\texttt{PASE\_Dec}(\widetilde{SK}, x_i)$. We differentiate between two cases for every x_i:

1. x_i is not leaf: $\mathcal{S}$ reads the id of x_i and searches the corresponding y_i in $\mathcal{X}(\mathbf{s}, T, R, t)$. It returns a pointer to all children in the order defined by t.
   $\mathcal{A}$ cannot distinguish between the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and the simulated output, because the pointers point to indistinguishable nodes according to the IND-CCA security of PASE. Furthermore, the results are consistent for different requests of the same range as the nodes access pattern delivers deterministic results and the pseudorandom permutation creates unambiguous positions for the simulated nodes. The same argument applies for queries of distinct or overlapping ranges.
2. x is leaf: $\mathcal{S}$ uses the leakage $\Delta(\mathbf{s}, T, R, t)$ to output all result pointers $\mathbf{P} = \bigcup_x \mathbf{P}_x, \forall(x, \mathbf{P}_x) \in \Delta$ in the order defined by t.
   This output is indistinguishable from the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$ as the number of result pointers matches and the pointers are consistent because $\Delta(\mathbf{s}, T, R, t)$ is unambiguous. The values pointed on are indistinguishable, because they are protected by IND-CCA secure encryption.

□

C Multiple Users

So far, we considered a setup comprising one user, but multiple user are directly supported by HardIDX. Multiple users are able to concurrently query data without limitations, as concurrent tree traversals do not influence each other. The only requirement is that each user has access to the key SK_k to create query tokens and SK_v to decrypt the result. It is also possible that each user shares a different key SK_k with the enclave. This would hide the search pattern of one user from all other users, but it requires a small modification in the protocol: the token has to be accompanied by client information, because the enclave has to identify the key to use for the token decryption. The nodes can be encrypted by any key that is known to the enclave. Particularly, it is not required to be a key shared with any user.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: ACM International Conference on Management of Data. SIGMOD (2004)
2. Anati, I., Gueron, S., Johnson, S.P., Scarlata, V.R.: Innovative Technology for CPU Based Attestation and Sealing. In: Workshop on Hardware and Architectural Support for Security and Privacy. HASP (2013)
3. ARM Limited: ARM Security Technology – Building a Secure System using TrustZone Technology (2009)
4. Bajaj, S., Sion, R.: TrustedDB: A trusted hardware-based database with privacy and data confidentiality. IEEE Transactions on Information Forensics and Security (2014)
5. Baumann, A., Peinado, M., Hunt, G.: Shielding Applications from an Untrusted Cloud with Haven. In: 11th USENIX Symposium on Operating Systems Design and Implementation. OSDI (2014)
6. Bellare, M., Boldyreva, A., O'Neill, A.: Deterministic and efficiently searchable encryption. In: 27th International Conference on Advances in Cryptology. CRYPTO (2007)

7. Bernstein, D.J., Lange, T., Schwabe, P.: The Security Impact of a New Cryptographic Library. In: 2nd International Conference on Cryptology and Information Security in Latin America. LATINCRYPT (2012)
8. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A.: Order-preserving symmetric encryption. In: 28th International Conference on Advances in Cryptology. EUROCRYPT (2009)
9. Boldyreva, A., Chenette, N., O'Neill, A.: Order-preserving encryption revisited: improved security analysis and alternative solutions. In: 31st International Conference on Advances in Cryptology. CRYPTO (2011)
10. Boneh, D., Sahai, A., Waters, B.: Functional encryption: definitions and challenges. In: 8th Conference on Theory of Cryptography. TCC (2011)
11. Boneh, D., Waters, B.: Conjunctive, subset, and range queries on encrypted data. In: 4th Theory of Cryptography Conference. TCC (2007)
12. Brasser, F., El Mahjoub, B., Koeberl, P., Sadeghi, A.R., Wachsmann, C.: TyTAN: Tiny Trust Anchor for Tiny Devices. In: Design Automation Conference. DAC (2015)
13. Brumley, B.B., Tuveri, N.: Remote Timing Attacks Are Still Practical. ESORICS, Springer Berlin Heidelberg (2011)
14. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: 33rd International Conference on Advances in Cryptology. CRYPTO (2013)
15. Costan, V., Devadas, S.: Intel SGX Explained. Tech. rep., IACR Cryptology ePrint Archive, Report 2016/086
16. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: 13th ACM Conference on Computer and Communications Security. CCS (2006)
17. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.: Practical Private Range Search Revisited. In: International Conference on Management of Data. SIGMOD (2016)
18. Durak, F.B., DuBuisson, T.M., Cash, D.: What else is revealed by order-revealing encryption? In: Conference on Computer and Communications Security. CCS (2016)
19. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: Beyond exact matches. In: 20th European Symposium on Research in Computer Security. ESORICS (2015)
20. Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., Sadeghi, A.R.: HardIDX: Practical and Secure Index with SGX (2017)
21. Garg, S., Mohassel, P., Papamanthou, C.: TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In: 36th Cryptology Conference. CRYPTO (2016)
22. Genkin, D., Pipman, I., Tromer, E.: Get your hands off my laptop: physical side-channel key-extraction attacks on PCs. Journal of Cryptographic Engineering (2015)
23. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Symposium on Theory of Computing. STOC (2009)
24. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: 32nd International Conference on Advances in Cryptology. CRYPTO (2012)
25. Grubbs, P., Sekniqi, K., Bindschaedler, V., Naveed, M., Ristenpart, T.: Leakage-Abuse Attacks against Order-Revealing Encryption. Tech. rep., IACR Cryptology ePrint Archive, Report 2016/895
26. Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., Del Cuvillo, J.: Using Innovative Instructions to Create Trustworthy Software Solutions. In: Workshop on Hardware and Architectural Support for Security and Privacy. HASP (2013)
27. Kaplan, D., Powell, J., Woller, T.: AMD Memory Encryption (2016)

28. Kerschbaum, F., Schröpfer, A.: Optimal average-complexity ideal-security order-preserving encryption. In: 21st ACM Conference on Computer and Communications Security. CCS (2014)
29. Lu, Y.: Privacy-preserving logarithmic-time search on encrypted data in cloud. In: 19th Network and Distributed System Security Symposium. NDSS (2012)
30. Maheshwari, U., Vingralek, R., Shapiro, W.: How to Build a Trusted Database System on Untrusted Storage. In: 4th Conference on Symposium on Operating System Design & Implementation. OSDI (2000)
31. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative Instructions and Software Model for Isolated Execution. In: Workshop on Hardware and Architectural Support for Security and Privacy. HASP (2013)
32. Mowery, K., Keelveedhi, S., Shacham, H.: Are AES x86 Cache Timing Attacks Still Feasible? In: ACM Workshop on Cloud Computing Security Workshop. CCSW (2012)
33. Naveed, M.: The fallacy of composition of oblivious RAM and searchable encryption. Tech. rep., IACR Cryptology ePrint Archive, Report 2015/668
34. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: 22nd ACM Conference on Computer and Communications Security. CCS (2015)
35. Ohrimenko, O., Schuster, F., Fournet, C., Meht, A., Nowozin, S., Vaswani, K., Costa, M.: Oblivious Multi-Party Machine Learning on Trusted Processors. In: 25th USENIX Security Symposium. USENIX Security (2016)
36. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H.: CryptDB: protecting confidentiality with encrypted query processing. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles. SOSP (2011)
37. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd Edition. McGraw-Hill, 3rd edn. (2002)
38. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: VC3: Trustworthy Data Analytics in the Cloud Using SGX. In: IEEE Symposium on Security and Privacy. S&P (2015)
39. Shen, E., Shi, E., Waters, B.: Predicate privacy in encryption systems. In: 6th Theory of Cryptography Conference. TCC (2009)
40. Shi, E., Bethencourt, J., Chan, H.T.H., Song, D.X., Perrig, A.: Multi-dimensional range query over encrypted data. In: IEEE Symposium on Security and Privacy. S&P (2007)
41. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy. S&P (2000)
42. Strackx, R., Piessens, F., Preneel, B.: Efficient Isolation of Trusted Subsystems in Embedded Systems. In: SecureComm (2010)
43. Wang, B., Hou, Y., Li, M., Wang, H., Li, H.: Maple: Scalable Multi-dimensional Range Search over Encrypted Cloud Data with Tree-based Index. In: 9th ACM Symposium on Information, Computer and Communications Security. ASIACCS (2014)
44. Xu, Y., Cui, W., Peinado, M.: Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: IEEE Symposium on Security and Privacy. S&P (2015)
45. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: 23rd USENIX Security Symposium. USENIX Security (2014)