A Secure One-Roundtrip Index for Range Queries

Tobias Boelter Rishabh Poddar Raluca Ada Popa
UC Berkeley

Abstract

We present the first one-roundtrip protocol for performing range, range-aggregate, and order-by-limit queries over encrypted data, that both provides semantic security and is efficient. We accomplish this task by chaining garbled circuits over a search tree, using *branch-chained garbled circuits*, as well as carefully designing garbled circuits. We then show how to build a database index that can answer order comparison queries. We implemented and evaluated our index. We demonstrate that queries as well as inserts and updates are efficient, and that our index outperforms previous interactive constructions. This index is part of the Arx database system, whose source code will be released in the near future.

1 Introduction

Operations that perform order comparison are crucial for databases. In many applications, such operations run on private or confidential fields such as grades, medical diagnosis, or timing information. TPC-C [TPC-C], an industry standard benchmark for databases, lists the following order operations as representative for a class of applications: range queries, order by, order by limit L (which returns the top L values), and aggregation on top of range queries (such as the sum over a range).

To protect sensitive data, there has been a significant amount of work on computing these operations on encrypted data. Despite many schemes proposed, none of them provide the functionality, security, and performance needed in a database setting.

A Desired Scheme for Databases. Databases use indexes to perform such comparison queries efficiently (e.g. in logarithmic time). Indexes are search trees in which the data is organized in sorted order. The server traverses the tree to answer the operations above. For example, for a query over the range $[a, b]$, the database server traverses the tree to find a and b, and then retrieves the values stored at the leaves between a and b. For an order by limit L query, the server retrieves the leftmost L items in the index or in a range.

A secure version of the index would enable the server to do similar work without learning the values stored in the index or the range $[a, b]$. Concretely, the encrypted index should provide semantic security. The index should not reveal anything about the data, other than the number of data items. In particular, the index should not reveal the ordering of the data in the database. When the server receives a query to perform an operation, the server should still be able to traverse the tree; the server sees the path in the tree it takes, but the server should not learn anything else about the data in the index or in the query. For a range query, once the server identified the database rows that match the range, the server should not also learn their order. We formalize the security of this index in §4.3.

To sum up the desired properties, we would like a database index that satisfies the following goals:

- *Functionality goal*: The index should support range and order-by-limit queries, as well as aggregates over these queries. It should also be possible to remove elements and insert new elements.

- *Performance goal*: The index should retain the logarithmic asymptotics of the regular database indexes as well as be fast in absolute terms. Concretely, the constants should be low enough that the index can be used in real applications.
- *Security goal*: The encrypted index should provide semantic security, formally defined in §4.3. The pattern of accesses observable to an attacker should be as above.

Contributions. In this paper, we construct a new database index, called the *reactive index*, that achieves all the properties above. The index is based on a chaining of garbled circuits we call *branch-chained garbled circuits* inspired from the Garbled RAM [GLO15] literature, as well as a careful design of garbled circuits. Since garbled circuits can be used at most once, we devise a fast repair procedure. We call our index scheme *reactive* because it reacts similarly to a chemical reaction upon receiving input: as we explain below, a part of the index combines with the input and gets consumed in the process of creating an output. For further use, parts of the index then need to be repaired.

Moreover, we show that this scheme implies a one-roundtrip order-preserving encoding scheme (without large client-side storage) not known before.

Finally, we show how to use the *reactive index* to execute database queries relying on order comparison. We designed and implemented the reactive index as part of the new Arx database system, which offers a rich set of operations in addition to order queries and shares a similar philosophy with this scheme.

Related Work. There is an extensive literature on computing range queries and order operations on encrypted data, with the goal to improve the tradeoff between functionality, security, and performance. However, no scheme achieves all the desired security, functionality, and performance goals we are aiming for. We survey this literature in §2.

Summary of Technique. For illustration purposes, we begin by describing a strawman construction that almost achieves the desired security. Consider a simple database of key-value pairs. Now consider the key-value pairs arranged in a binary search tree sorted by key where each node is IND-CPA encrypted by the client. Every time the server needs to search the tree for a particular value a, e.g., as part of processing a range query over $[a, b]$, the client helps the server traverse the tree. Whenever the server needs to decide if the encrypted value at a node v of the tree is smaller or larger than a, the server provides this node to the client, the client decrypts it, compares it to a, and tells the server the result of the comparison. To achieve the desired security (which we elaborate on in §4.3) also for inserts and deletes, we use a history-independent data structure for the tree, so that the shape of the tree does not depend on the order of the operations.

However, this strawman is an interactive protocol with as many rounds of interaction as levels in the tree. Considering a cloud servicing scenario with network latencies in the order of 5-50ms per roundtrip, the performance overhead of the additional roundtrip is significant. This technique was actually proposed in mOPE [PLZ13] to achieve order-preserving encryption and is also currently used in the database system ZeroDB [EW16] for range queries. In contrast, as we discuss in §10, our index processes an entire range query in less than a roundtrip time.

Inspired by work on Black-Box Garbled RAM [GLO15], we construct a scheme that uses garbled circuits organized in a tree to reduce interaction. The idea is that instead of storing IND-CPA ciphertexts at each node in the tree, the server stores a garbled circuit, each of which hardcodes the value at the node, say y, into a comparison circuit. Given input labels for a value x, the garbled circuit compares x to y, and outputs 1 if $x < y$ and 0 otherwise, which tells the server to go left or right respectively. Since the client cannot supply labels for the child garbled circuit (because the client does not know if the server will go left or right in advance), we also provide a mechanism to the server to convert output labels into input labels for the correct child. In this manner, the server can traverse the tree down to the leaf with no additional help from the client.

Since garbled circuits may only be used once, the client provides new garbled circuits to replace the used ones. Fortunately, the expected number of garbled circuits consumed is logarithmic. We call this the *repair* procedure.

We have put significant effort into making this technique practical. We make the garbled circuits small through a set of techniques such as a small comparison circuit, transition tables to encode output labels, using the half gates techniques of [ZRE15], and a compressed representation of these circuits.

Implication to Order-Preserving Encryption Schemes. We also show that our reactive index can be used to construct a *one-round ideal-security mutable order-preserving encoding scheme*, achieving both an ideal notion of security for such schemes and of asymptotic performance, the first such solution. [BCL+09] formalizes the ideal security for order-preserving schemes (IND-OCPA); intuitively, the server should learn nothing about the encrypted values other than their order. Regarding performance, the scheme should ideally not require more than one roundtrip between the client and the server to return an answer. Moreover, the state at the client should be small (e.g., logarithmic in the size of the database). A scheme that stores all the values encrypted at the client trivially achieves this property but is not desirable in practice.

So far, there is no order-preserving scheme that achieves both of these properties. We show that our reactive index can be used to implement such a scheme, the construction of which follows directly from the construction of our index.

Evaluation Results. We implemented our reactive index using Java and C++. The resulting size of each garbled circuit is small. Each node in our index needs to store $n \cdot 6 \cdot 128 + 128$ bits for the garbled circuit and transition tables, which for a bit-length of 32 bits results in a total of 3088 bytes. In comparison, a Paillier ciphertext whose only ability is to compute addition, requires 512 bytes. Furthermore, the time to evaluate a 32-bit garbled comparison circuit is 10839 cycles (0.0036ms) and computing a complete range query on a data-set of size 5 million is in the order of milliseconds.

2 Related Work

There is an extensive amount of work on computing order operations on encrypted data, with the goal to improve the tradeoff between functionality, security, and performance.

We organize related work in four categories: (1) order-preserving encryption and similar schemes, (2) functional encryption and similar range query schemes, (3) schemes derived from searchable encryption, and (4) other related work. The above categories are not mutually exclusive.

None of the categories above achieve all the desired security, functionality, and performance goals outlined in the introduction, §1. Table 1 summarizes their shortcomings. In the rest of the section, we explain each category in part.

2.1 Order-Preserving Encryption (OPE) and Similar Schemes

An encryption scheme Enc is order-preserving, if it preserves the order which can either mean $a < b \implies \mathsf{Enc}(a) < \mathsf{Enc}(b)$ or the stricter property $a \leq b \implies \mathsf{Enc}(a) \leq \mathsf{Enc}(b)$, making the encryption scheme effectively deterministic.

There has been a significant amount of work on OPE both in the research community [ÖSC03; AKS+04; YKK+11; AEE+09; LPL+09; KAK10; XYH12; LW12; LW13; PLZ13; BCL+09; BCO11] and in industry [AWW12; Per; Cip; Ras11]. The earlier constructions were ad-hoc and insecure. The first rigorous

Category of Related Work	**Functionality**	**Desired Security**	**Performance**
Order-preserving encryption and similar, §2.1	YES	**NO**	YES
Functional encryption schemes and similar, §2.2	**NO**	YES	**NO**
Searchable encryption-based schemes, §2.3	**NO**	**NO**	YES
Our reactive index	YES	YES	YES

Table 1: Related work comparison. The last three columns indicate if the related work meets the goals outlined in §1. In some categories, the tradeoff differs from work to work, so we considered the most representative works.

treatment was provided by [BCL⁺09]. This work included two security definitions: (1) indistinguishability from a random order preserving function (ROPF) and (2) indistinguishability under ordered chosen plaintext attack (IND-OCPA). They also provided an OPE scheme secure under the first definition. Follow-up work [BCO11] showed that the ROPF model is insufficient and that IND-OCPA is desired. [PLZ13] showed that IND-OCPA is impossible for a encryption scheme in the normal sense. They circumvent this impossibility result by relaxing the syntax of an encryption scheme and allow ciphertexts to be changed after encryption, and they provide the first IND-OCPA secure construction, called mOPE. Unfortunately, their encryption scheme requires interaction between the client who holds the private key, and the server who holds all previously generated ciphertexts. Subsequent work tried to reduce the number of interactions and to make the scheme randomized [KS14; Ker15].

Another way to circumvent the IND-OCPA impossibility is to consider order-revealing encryption. In order revealing encryption, there exists a public algorithm capable of comparing any two ciphertexts. There exists an ORE scheme based on multi-linear maps [BLR⁺14] that provides IND-OCPA but is due to the multi-linear maps to inefficient in practice. An efficient ORE scheme was proposed [CLW⁺16] but it leaks the first bit in which two ciphertexts differ which is significant.

Overall, these schemes deliver rich functionality including the one we target. At the same time, some schemes also deliver reasonable performance. But even an IND-OCPA secure encryption scheme reveals the order of ciphertexts by definition at rest. In many application domains this leakage already significantly compromises security. Specifically for applications with a small domain size like age or date, OPE and ORE are insecure [SW15; CGP⁺15]

2.2 Functional Encryption and Related Range Query Schemes

A set of schemes focus on performing range queries over encrypted data [HIL⁺02; SBC⁺07; BW07; Lu12]. With these schemes, one encrypts data values and query values with different algorithms, and the server can learn the order between a query value and a data value. The security goal is to not reveal anything else about the data, namely, no order relations between two query or two data values, which satisfies our desired security guarantee.

Unfortunately, no efficient constructions achieve this ideal goal; some schemes reveal all query values, other schemes reveal some data values, and some schemes provide only approximate answers.

On the theoretical side, Functional Encryption can generally be achieved under multi-linear map assumptions [GGH⁺16]. The special case of range predicate encryption was constructed in [Lu12] using inner product predicate encryption, yielding a scheme with reasonable performance for a small number of ciphertexts but still impractical for performing full database scans. Also, this scheme is not adaptively secure, which makes updates insecure. Although there are generic ways to compile selectively secure functional encryption into adaptively secure functional encryption [ABS⁺15], their practicality is uncertain.

Regarding functionality, these schemes do not support *order by* and *order by limit*, which are two basic operations required by databases.

2.3 Searchable Encryption-Based Schemes

Some recent proposals [FJK+15] build range query schemes from searchable encryption. At a high level, these schemes map a range $[a, b]$ to a set of prefixes, and then search for data values that match those prefixes by equality. The result of a query often contains more data values than belong to the range, and the client filters these values.

While these schemes support range queries, they do not support order by, order by limit, and aggregates over range queries. They do not support the latter because the result of the aggregate contains data values outside of the range as well and is thus not correct. Order by limit are often more common than range queries (e.g., TPC-C [TPC-C]) because they are used for pagination.

Regarding security, such schemes are comparable with our index scheme, and neither is more secure than the other. For example, these schemes leak more about the range being matched, e.g., the size of the range. For data values that match a range, the scheme additionally leaks if two values match a prefix. On the other hand, the access pattern of inserting values leaks less with these schemes than with our index. Developing a (practical) hybrid of these schemes and our index that has the best of both worlds is a useful open question.

2.4 Other Related Work

Our tree of garbled circuits is similar in spirit to Black-Box Garbled RAM [GLO15] and TWORAM [GMP15]. Unlike this work, our scheme is practical. We designed it specifically for our setting. In §1, we explained the various other techniques we contribute on top of this approach.

Secure Indexes [Goh03] are data structures that allow a querier with a "trapdoor" for a word x to test in $\mathcal{O}(1)$ time only if the index contains x.

3 Notation

Let λ denote the security parameter throughout this paper. For a distribution $\mathcal{D}$, we say $x \leftarrow \mathcal{D}$ when x is sampled from the distribution $\mathcal{D}$. If S is a finite set, by $x \leftarrow S$ we mean x is sampled from the uniform distribution over the set S. We use $p(\cdot)$ to denote that p is a function that takes one input. Similarly, $p(\cdot, \cdot)$ denotes a function p that takes two inputs.

We say that a function f is negligible in an input parameter λ, if for all $d > 0$, there exists K such that for all $\lambda > K$, $f(\lambda) < k^{-d}$. For brevity, we write: for all sufficiently large λ, $f(\lambda) = \mathrm{negl}(\lambda)$. We say that a function f is polynomial in an input parameter λ, if there exists a polynomial p such that for all λ, $f(\lambda) \leq p(\lambda)$. We write $f(\lambda) = \mathrm{poly}(\lambda)$. A similar definition holds for $\mathrm{polylog}(\lambda)$.

Two ensembles, $X = \{X_\lambda\}_{\lambda \in \mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda \in \mathbb{N}}$, are said to be *computationally indistinguishable* (and denoted $\{X_\lambda\}_{\lambda \in \mathbb{N}} \overset{c}{\approx} \{Y_\lambda\}_{\lambda \in \mathbb{N}}$) if for every probabilistic polynomial-time algorithm D,

$$|\Pr[D(X_\lambda, 1^\lambda) = 1] - \Pr[D(Y_\lambda, 1^\lambda) = 1]| = \mathrm{negl}(\lambda).$$

4 Definition of a Secure Reactive Index

4.1 Overview

A reactive index is an interactive protocol executed between a stateful client and a stateful server. In our definition and construction, we model the server as an honest-but-curious attacker. We then discuss a straightforward extension to protecting against a malicious server-side attacker in §11.

At setup time, our scheme allows bulk-loading an initial database; the database can also be modified through subsequent queries. The bulk-load functionality models a read-only database and is more efficient than creating the database by inserting one element at a time.

The database is modeled as a list of key-value pairs. Keys do not need to be unique. This model suffices for a more complex relational model with many columns per table because each value can store multiple data items or can be a pointer to the whole document/record associated with this key. In Arx, the value is both a pointer to values in a table row and some precomputed values.

When the client issues a query, the query is encrypted at the client and sent to the server who runs this query on the encrypted data to produce an encrypted answer. The server can answer with the response in full, requiring only one roundtrip.

The execution of the query "damages" the index partially. To fix the index, the client needs to provide *repair* information. As we will see, this repair information is small and can be provided with the next query, not requiring a separate roundtrip.

We continue with the formal definition.

4.2 Formal Definition

Remark 4.1. *Throughout the paper we assume algorithms to be random access machines. As a result, an algorithm's runtime may be lower than the size of the memory it is working on but the output can still consist of the whole memory.*

Definition 4.1 (Database)**.** *A database $\mathcal{DB}$ is a list of values of which each is associated with a not necessarily unique key: $\mathcal{DB} \subset \mathbb{Z} \times \{0,1\}^*$*

Definition 4.2 (Query Functionality)**.** *The functionality Func_q of a query q is a polynomial time algorithm $(\mathcal{DB}', \mathcal{R}) \leftarrow \mathsf{Func}_q(\mathcal{DB})$ that on input a database $\mathcal{DB}$ outputs an updated database DB' as well as a result set comprising of key-value pairs.*

Definition 4.3 (Queries)**.**

1. *For $q = (\mathit{RANGE}, [a,b])$, $\mathsf{Func}_q(\mathcal{DB})$ outputs $(\mathcal{DB}, \mathcal{R}_{[a,b]} = \{(k,v) : (k,v) \in \mathcal{DB}, k \in [a,b]\})$.*

2. *For $q = (\mathtt{ORDER\ LIMIT}, [a,b], \text{start}, \text{length})$, let L be the list of elements in $\mathcal{R}_{[a,b]}$ ordered by k. Then $\mathsf{Func}_q(\mathcal{DB})$ outputs $(\mathcal{DB}, (L[\text{start}], \ldots, L[\text{start} + \text{length}])$.*

3. *For $q = (\mathit{AGGREGATE}, [a,b])$, $\mathsf{Func}_q(\mathcal{DB})$ outputs*

$$(\mathcal{DB}, \sum_{(k,v) \in \mathcal{R}_{[a,b]}} f(k,v))).$$

   Where f can be any fixed function that returns an integer and $+$ is a fixed associative operator. For ease of discourse, we assume this operator to be plain addition in the remainder of the paper. The generalization is straight-forward.

4. *For* $q = (\mathit{INSERT}, k, v)$*,* $\mathsf{Func}_q(\mathcal{DB})$ *outputs* $(\mathcal{DB}' = \mathcal{DB}.\mathrm{append}(k, v)), \emptyset)$*.*

5. *For* $q = (\mathit{DELETE}, [a, b])$*,* $\mathsf{Func}_q(\mathcal{DB})$ *outputs* $(\mathcal{DB}' = \{(k, v) : (k, v) \in \mathcal{DB}, k \notin [a, b]\}, \emptyset)$*.*

For a database $\mathcal{DB}$ *we write short* $\mathcal{DB}(q_1, \ldots, q_n)$ *for the database that results by applying iteratively* Func_{q_i} *to* $\mathcal{DB}$*.*

Definition 4.4 (Syntax of a Reactive Index)**.** *A reactive index scheme* Index = (ClientSetup, EncQuery, RunQuery, DecResult, Repair) *is a tuple of* PPT *algorithms executed by a client and a server with the following properties:*

$(\mathcal{I}, \mathcal{S}) \leftarrow \mathsf{ClientSetup}(1^\lambda, \mathcal{DB})$ *is executed by the client and, on input the security parameter* λ *in unary representation and an initial database, outputs private internal state* $\mathcal{S}$ *which the client will use in the later execution of the protocol as well as the encrypted index* $\mathcal{I}$*.*

$\overline{\mathcal{Q}} \leftarrow \mathsf{EncQuery}(\mathcal{S}, q)$ *is executed by the client and on input the private internal state* $\mathcal{S}$ *and a query* q*, outputs an encrypted query* $\overline{\mathcal{Q}}$*.*

$(\overline{\mathcal{R}}, \mathcal{I}') \leftarrow \mathsf{RunQuery}(\mathcal{I}, \overline{\mathcal{Q}})$ *is executed by the server and on input the index and an encrypted query, outputs an encrypted result as well as an updated index.*

$(\mathcal{R}, \mathcal{H}, \mathcal{S}') \leftarrow \mathsf{DecResult}(\mathcal{S}, \overline{\mathcal{R}})$ *is executed by the client, and on input* $\mathcal{S}$ *and an encrypted result* $\overline{\mathcal{R}}$*, outputs the result set in clear text, repair data* $\mathcal{H}$ *and an updated secret state.*

$(\mathcal{I}') \leftarrow \mathsf{Repair}(\mathcal{I}, \mathcal{H})$ *is executed by the server, and on input the index and repair data, outputs an updated index.*

Remark 4.2 (Round Complexity)**.** *The round complexity in the definition of the reactive index is 4, i.e. two roundtrips: One for the query and one for the repair. This notation makes specifying and analyzing the security guarantees easier but in practice this behaves like a one-roundtrip protocol because the result is already obtained after the first roundtrip and the repair phase can be integrated into the following query phase.*

In order to say a reactive index is correct, we require that all queries executed in a standard database return the same result when they are executed using the reactive index.

Definition 4.5 (Correctness)**.** *A reactive index* Index *supports a set of queries* $\mathfrak{Q}$ *if for any sequence of queries* $(q_1, \ldots, q_k)$*,* $q_i \in \mathfrak{Q}$*, and any initial database* $\mathcal{DB}_0$*, the results of the queries when executed on plain text are the same as when executed with* Index*. Concretely, for* $i = 1, \ldots, k$ *let*

$$\begin{aligned}
(\mathcal{DB}_i, \mathcal{R}_i^\times) &\leftarrow \mathsf{Func}_{q_i}(\mathcal{DB}_{i-1}) \\
(\mathcal{I}_0, \mathcal{S}_0) &\leftarrow \mathsf{ClientSetup}(1^\lambda, \mathcal{DB}_0) \\
\overline{\mathcal{Q}}_i &\leftarrow \mathsf{EncQuery}(\mathcal{S}_{i-1}, q_i) \\
(\overline{\mathcal{R}}_i, \mathcal{I}_i) &\leftarrow \mathsf{RunQuery}(\mathcal{I}_{i-1}, \overline{\mathcal{Q}}_i) \\
(\mathcal{R}_i, \mathcal{H}_i, \mathcal{S}_i) &\leftarrow \mathsf{DecResult}(\mathcal{S}_{i-1}, \overline{\mathcal{R}}_i) \\
(\mathcal{I}_i) &\leftarrow \mathsf{Repair}(\mathcal{I}_i, \mathcal{H}_i)
\end{aligned}$$

then we require that

$$\mathcal{R}_i = \mathcal{R}_i^\times .$$

A correct and secure index could so far easily achieved by storing the whole database on the client, which defeats the purpose of our definition. Therefore we have efficiency requirements, which are concisely formulated in the following definition. All other expected efficiency requirements follow from the requirement that all algorithms are PPT.

Definition 4.6 (Efficiency)**.** *A reactive index* Index *is efficient, if the expected size of the client state and repair data* $|\mathcal{H}| = |\mathcal{S}| = \mathcal{O}(\lambda \log |\mathcal{DB}|)$ *at all times where* λ *is the security parameter. And the runtime of* $\mathsf{RunQuery}(\mathcal{I}, \overline{\mathcal{Q}}) = \mathcal{O}(|\mathcal{R}| + \log |\mathcal{I}|)$ *in the random access machine model.*

4.3 Threat Model and Security Definition

Our model consists of two parties, clients and a server, and corresponds to a cloud outsourcing model. The clients are trusted. We present our schemes for an *honest but curious* attacker at the server, but discuss an extension to a malicious attacker. The server attackers we consider are powerful. Such attackers include curious employees of a cloud provider, hackers breaking into the database servers, or subpoenas. The attackers can see *all* the information at the server: the entire contents of the database, any data or keys stored in memory, and any network messages received by the server. Hence, if the decryption key is in main memory, the attacker can reach it and decrypt the database.

To better undestand our security guarantees, we categorize the attackers into two kinds: snapshot and persistent attackers. The snapshot attacker steals a snapshot of the database (tables and indices included). The persistent attacker manages to install a logger that records over time the accesses and cryptographic tokens from the client and then sends it to the attacker.

The snapshot attacker is by far the most common attacker we encounter today. Hackers or insiders simply steal versions of the database. For this attacker, the database in our scheme is encrypted with IND-CPA security, providing indistinguishability under chosen plaintext attack.

For the persistent attacker, we formulate our security guarantees using a standard leakage function and prove semantic security up to this leakage. We note that the persistent attacker is less common than the snapshot attacker and is much easier to detect and stop than a snapshot attacker. The longer the attacker stays in the system, the more likely it gets caught. Compared to a snapshot attacker, a persistent attacker gets to see side channel information: timing attacks (e.g., the time when a query arrives indicates the user is online) or attacks based on access patterns (which positions in the index are accessed and how frequently, but not their contents). This attacker still does not see index contents as before. Our index scheme does not prevent such leakage. Oblivious RAM [SSS12] is an active area of research that hides access patterns, and Garbled RAM [GLO15] enables combining Oblivious RAM with computation like in our index. Despite significant progress on these fronts in recent years, there still does not exist a cost-effective solution.

Because we want security of previous queries for an attacker who breaks in at a later point, we need to leverage the secure erasure model where the honest server is assumed to be able to securely erase data without any trace.

Next, we give a security definition based on the Real/Ideal paradigm that captures the above threat model. The definition uses a leakage function that precisely models what an adversary should be able to learn from observing one or more queries to the database.

Definition 4.7 (Leakage, Reproducible Leakage)**.** *For a reactive index* Index *we define a leakage function* $\mathsf{Leak}(\mathcal{DB})$*,* $\mathsf{Leak}(\mathcal{DB}, q)$ *that maps a databases* $\mathcal{DB}$ *and database-query pairs to strings formalizing the information the server is allowed to learn in the protocol. We write* $\mathsf{Leak}(q)$ *short for* $\mathsf{Leak}(\mathcal{DB}, q)$ *if the database is known from the context. We call a string* $x = (x_0, \ldots, x_n)$ reproducible leakage*, if there exists a* *database $\mathcal{DB}$ and queries $(q_1, \ldots, q_n)$ such that $x = (\mathsf{Leak}(\mathcal{DB}), \mathsf{Leak}(q_1), \ldots, \mathsf{Leak}(q_n))$ and this database and queries are efficiently computable.*

We provide both adaptive and non-adaptive security definitions. We will prove security only for the non-adaptive case. This suffices for our setting because the attacker does not control the queries issued by the client and the clients issue these queries without seeing the encrypted data. Nevertheless, we are working on an adaptive security proof, but this requires complicated modifications to our scheme.

Definition 4.8 (Adaptive Semantic Security)**.** *Let* Index = (ClientSetup, EncQuery, RunQuery, DecResult, Repair) *be a reactive index,* $\mathsf{Adv} = (\mathsf{Adv}_1, \ldots, \mathsf{Adv}_{n+3})$ *an adversary and* $\mathsf{Sim} = (\mathsf{Sim}_1, \ldots, \mathsf{Sim}_{n+3})$ *a simulator. We consider the following two experiments* $(Real, Ideal)$*.*

$\mathbf{Real}_{\mathsf{Index},\mathsf{Adv}}(1^\lambda)$

```
$(\mathcal{A}, \mathcal{DB}, (q_1, \ldots, q_m)) \leftarrow \mathsf{Adv}_1(1^\lambda)$. The adversary generates an initial database, a list of $m$ initial queries, and outputs its internal state.
$(\mathcal{I}_0, \mathcal{S}_0) \leftarrow \mathsf{ClientSetup}(1^\lambda, \mathcal{DB})$
for $i = 1, \ldots, m$ do
    $\overline{\mathcal{Q}}_i \leftarrow \mathsf{EncQuery}(\mathcal{S}_{i-1}, q_i)$
    $(\overline{\mathcal{R}}_i, \mathcal{I}_i) \leftarrow \mathsf{RunQuery}(\mathcal{I}_{i-1}, \overline{\mathcal{Q}}_i)$
    $(\mathcal{R}_i, \mathcal{H}_i, \mathcal{S}_i) \leftarrow \mathsf{DecResult}(\mathcal{S}_{i-1}, \overline{\mathcal{R}}_i)$
    $\mathcal{I}_i \leftarrow \mathsf{Repair}(\mathcal{I}_i, \mathcal{H}_i)$

$\mathcal{A} \leftarrow \mathsf{Adv}_2(\mathcal{I}_n, \mathcal{A})$. The adversary learns the initial dump of the database at the time it breaks in.
for $i = m+1, \ldots, m+n$ do
    $(\mathcal{A}, q_i) \leftarrow \mathsf{Adv}_{i-m+2}(\mathcal{A})$. The adversary chooses a query.
    if $q_i \notin \mathfrak{Q}$ then
        break
    $\overline{\mathcal{Q}}_i \leftarrow \mathsf{EncQuery}(\mathcal{S}_{i-1}, q_i)$


    $(\overline{\mathcal{R}}_i, \mathcal{I}_i) \leftarrow \mathsf{RunQuery}(\mathcal{I}_{i-1}, \overline{\mathcal{Q}}_i)$
    $(\mathcal{R}_i, \mathcal{H}_i, \mathcal{S}_i) \leftarrow \mathsf{DecResult}(\mathcal{S}_{i-1}, \overline{\mathcal{R}}_i)$
    $\mathcal{I}_i \leftarrow \mathsf{Repair}(\mathcal{I}_i, \mathcal{H}_i)$
    $\mathcal{A} \leftarrow \mathsf{Adv}_{i-m+2}((\overline{\mathcal{Q}}_i, \mathcal{H}_i), \mathcal{A})$
output $x \leftarrow \mathsf{Adv}_{n+3}(\mathcal{A})$
```

$\mathbf{Ideal}_{\mathsf{Index},\mathsf{Adv},\mathsf{Sim},\mathsf{Leak}}(1^\lambda)$

```
$(\mathcal{A}, \mathcal{DB}, (q_1, \ldots, q_m)) \leftarrow \mathsf{Adv}_1(1^\lambda)$. The adversary generates an initial database, a list of $m$ initial queries, and outputs its internal state.

for $i = 1, \ldots, m$ do
    $(\mathcal{DB}, \mathcal{R}) \leftarrow \mathsf{Func}_{q_i}(\mathcal{DB})$



$\mathcal{B} \leftarrow \mathsf{Sim}_1(\mathsf{Leak}(\mathcal{DB}))$
$\mathcal{A} \leftarrow \mathsf{Adv}_2(\mathsf{Sim}_2(\mathcal{B}), \mathcal{A})$. The adversary learns the initial dump of the database at the time it breaks in.
for $i = m+1, \ldots, m+n$ do
    $(\mathcal{A}, q_i) \leftarrow \mathsf{Adv}_{i-m+2}(\mathcal{A})$. The adversary chooses a query.
    if $q_i \notin \mathfrak{Q}$ then
        break
    $(\mathcal{B}, \mathcal{D}) \leftarrow \mathsf{Sim}_{i-m+2}(\mathcal{B}, \mathsf{Leak}(\mathcal{DB}, q_i))$. The Simulator outputs fake data $\mathcal{D}$ and internal state $\mathcal{B}$.
    $(\mathcal{DB}, \mathcal{R}) \leftarrow \mathsf{Func}_{q_i}(\mathcal{DB})$


    $\mathcal{A} \leftarrow \mathsf{Adv}_{i-m+2}(\mathcal{D}, \mathcal{A})$
output $x \leftarrow \mathsf{Sim}_{n+3}(\mathcal{A})$
```

We call the reactive index Index adaptively semantic secure *with leakage* Leak, *if for all algorithms* $\mathsf{Adv} = (\mathsf{Adv}_1, \ldots, \mathsf{Adv}_{n+3})$ *for* $n = \mathrm{poly}(\lambda)$ *that are* $\mathrm{poly}(\lambda)$ *size, there exists a simulator* $\mathsf{Sim} = (\mathsf{Sim}_1, \ldots, \mathsf{Sim}_{n+3})$ *that is* $\mathrm{poly}(\lambda)$ *size such that* $\mathbf{Ideal}_{\mathsf{Index},\mathsf{Adv},\mathsf{Sim},\mathsf{Leak}}(1^\lambda)$ *with probability at most* $1/2$ *outputs a special symbol* $\perp$*, i.e.*

$$\Pr[\mathbf{Ideal}_{\mathsf{Index},\mathsf{Adv},\mathsf{Sim},\mathsf{Leak}}(1^\lambda) = \perp] \leq 1/2$$

and further, let $\mathbf{Ideal}^*$ *be the random variable describing* $\mathbf{Ideal}$ *conditioned on* $\mathbf{Ideal} \neq \bot$*, i.e.* $\forall \alpha \in \{0,1\}^*$:

$$\Pr[\mathbf{Ideal}^*(1^\lambda) = \alpha] = \Pr[\mathbf{Ideal}_{\mathsf{Index,Adv,Sim,Leak}}(1^\lambda) = \alpha \mid \mathbf{Ideal}_{\mathsf{Index,Adv,Sim,Leak}}(1^\lambda) \neq \bot]$$

Then

$$\mathbf{Real}_{\mathsf{Index,Adv}}(1^\lambda) \overset{c}{\approx} \mathbf{Ideal}^*(1^\lambda)$$

Definition 4.9 (Non-Adaptive Semantic Security)**.** *We call a reactive index* $\mathcal{I}$ *non-adaptively semantic secure with leakage* Leak*, if it satisfies definition 4.8 with the adjustment that the adversary* Adv_1 *choses not only* $(q_1, \ldots, q_m)$ *but also* $(q_{m+1}, \ldots, q_{m+n})$ *non-adaptively in advance.*

Definition 4.10 (Adaptive Indistinguishability)**.** *Let* Index = (ClientSetup, EncQuery, RunQuery, DecResult, Repair) *be a reactive index,* $\mathsf{Adv} = (\mathsf{Adv}_1, \ldots, \mathsf{Adv}_{n+3})$ *with* $n \in \mathbb{N}$*. We consider the following experiment.*

```
$\mathbf{Ind}_{\mathsf{Index,Adv,Leak}}(1^\lambda)$

$b \leftarrow \{0,1\}$. Chose a random bit $b$.
$(\mathcal{A}, \mathcal{DB}^0, (q_1^0, \ldots, q_m^0), \mathcal{DB}^1, (q_1^1, \ldots, q_m^1)) \leftarrow \mathsf{Adv}_1(1^\lambda)$. The adversary generates two initial
database, two lists of $m$ initial queries, and outputs its internal state.
if $\mathsf{Leak}(\mathcal{DB}^0(q_1^0, \ldots, q_m^0)) \neq \mathsf{Leak}(\mathcal{DB}^1(q_1^1, \ldots, q_m^1))$ then
    output 0 and abort.
$(\mathcal{I}_0, \mathcal{S}_0) \leftarrow \mathsf{ClientSetup}(1^\lambda, \mathcal{DB}^\mathsf{b})$
for $i = 1, \ldots, m$ do
    $\overline{\mathcal{Q}}_i \leftarrow \mathsf{EncQuery}(\mathcal{S}_{i-1}, q_i^b)$
    $(\overline{\mathcal{R}}_i, \mathcal{I}_i) \leftarrow \mathsf{RunQuery}(\mathcal{I}_{i-1}, \overline{\mathcal{Q}}_i)$
    $(\mathcal{R}_i, \mathcal{H}_i, \mathcal{S}_i) \leftarrow \mathsf{DecResult}(\mathcal{S}_{i-1}, \overline{\mathcal{R}}_i)$
    $\mathcal{I}_i \leftarrow \mathsf{Repair}(\mathcal{I}_i, \mathcal{H}_i)$
$\mathcal{A} \leftarrow \mathsf{Adv}_1(\mathcal{I}_n, \mathcal{A})$. The adversary learns the initial dump of the database at the time it breaks in.
for $i = m+1, \ldots, m+n$ do
    $(\mathcal{A}, q_i^0, q_i^1) \leftarrow \mathsf{Adv}_{i-m+2}(\mathcal{A})$. The adversary chooses two queries.
    if $\mathsf{Leak}(q_i^0) \neq \mathsf{Leak}(q_i^1)$ then output 0 and abort.
    $\overline{\mathcal{Q}}_i \leftarrow \mathsf{EncQuery}(\mathcal{S}_{i-1}, q_i^b)$
    $(\overline{\mathcal{R}}_i, \mathcal{I}_i) \leftarrow \mathsf{RunQuery}(\mathcal{I}_{i-1}, \overline{\mathcal{Q}}_i)$
    $(\mathcal{R}_i, \mathcal{H}_i, \mathcal{S}_i) \leftarrow \mathsf{DecResult}(\mathcal{S}_{i-1}, \overline{\mathcal{R}}_i)$
    $\mathcal{I}_i \leftarrow \mathsf{Repair}(\mathcal{I}_i, \mathcal{H}_i)$
    $\mathcal{A} \leftarrow \mathsf{Adv}_{i-m+2}((\overline{\mathcal{Q}}_i, \mathcal{H}_i), \mathcal{A})$
$b' \leftarrow \mathsf{Adv}_{n+3}(\mathcal{A})$
if $b = b'$ then
    output 1
else
    output 0
```

In the case that the experiment outputs 1*, we say* Adv wins *the experiment. We call the reactive index* Index adaptively secure in the indistinguishability sense *with leakage* Leak*, if for all algorithms* $\mathsf{Adv} = (\mathsf{Adv}_1, \ldots, \mathsf{Adv}_{n+3})$ *for* $n = \mathrm{poly}(\lambda)$ *that are* $\mathrm{poly}(\lambda)$ *size, there exists a negligible function* negl *such that*

$$\Pr[\mathbf{Ind}_{\mathsf{Index,Adv,Leak}}(1^\lambda) = 1] \leq 1/2 + \mathrm{negl}(\lambda)$$

Definition 4.11 (Non-Adaptive Indistinguishability)**.** *We call a reactive index $\mathcal{I}$ non-adaptively secure in the indistinguishability sense with leakage* Leak *if it satisfies definition 4.10 with the adjustment that* Adv *has to choose all queries non-adaptively in advance.*

Theorem 4.3. *Non-adaptive indistinguishability with reproducible leakage implies non-adaptive semantic security with the same leakage.*

Proof. Let Index be an reactive index that is non-adaptively secure in the sense of indistinguishability with leakage Leak. Given a adversary Adv, we construct a simulator Sim that, given oracle access to Adv simulates the ideal world (black-box simulation), i.e.

$$\mathbf{Real}_{\mathsf{Index},\mathsf{Adv}}(1^\lambda) \overset{c}{\approx} \mathbf{Ideal}_{\mathsf{Index},\mathsf{Adv},\mathsf{Sim},\mathsf{Leak}}(1^\lambda)$$

Intuitively, we let the simulator execute the real world protocol "in its head" with a database and list of queries that produces the same leakage as the database and queries chosen by Adv. If the output of this simulation happened to be distinguishable from the real world, this would effectively give us a distinguisher capable of winning the $\mathbf{Ind}$ experiment, violating the fact that Index is non-adaptively secure in the sense of indistinguishability.

Concretely, the simulator Sim, given the leakage of the adversary-chosen database $\mathcal{DB}$ and queries $(q_{m+1}, \ldots, q_{m+n})$, can reconstruct a database $\mathcal{DB}'$ and queries $(q'_{m+1}, \ldots, q'_{m+n})$, which are likely not the same, but share the same leakage. Following this step, Sim executes the same protocol but on this data instead of the adversary-supplied data. Hence, the adversaries view in $\mathbf{Real}_{\mathsf{Index},\mathsf{Adv}}(1^\lambda)$ is the execution for database $\mathcal{DB}$ and queries $(q_{m+1}, \ldots, q_{m+n})$ while in $\mathbf{Ideal}_{\mathsf{Index},\mathsf{Adv},\mathsf{Sim},\mathsf{Leak}}(1^\lambda)$ his view is the execution for database $\mathcal{DB}'$ and queries $(q'_{m+1}, \ldots, q'_{m+n})$. If the adversary's views are computationally indistinguishable, their output in the experiments is as well. Assume towards the contradiction that there exists a distinguisher $\mathcal{D}$ with non-negligible advantage. We use $\mathcal{D}$ to construct an adversary Adv^* to attack the experiment $\mathbf{Ind}_{\mathsf{Index},\mathsf{Adv},\mathsf{Leak}}(1^\lambda)$. Adv^* outputs $\mathcal{DB}$ and queries $(q_{m+1}, \ldots, q_{m+n})$ for the 0-slots and $\mathcal{DB}'$ and queries $(q'_{m+1}, \ldots, q'_{m+n})$ for the 1-slots in the execution of the experiment. At the end it runs $\mathcal{D}$ on its view and outputs its guess b' consistently with the output of $\mathcal{D}$. The advantage of Adv^* is exactly the advantage of $\mathcal{D}$, leading to the contradiction that Index is secure in the sense of indistinguishability. $\square$

Remark 4.4. *It seems like the definition of adaptive semantic security is strictly stronger than adaptive semantic indistinguishability. This is because in order to generate a database and a set of queries that produce the same leakage as another database and set of queries, all of them have to be known in advance. Otherwise the adversary will always able to, in fact very easily, adaptively trap the simulator into a situation where the simulator will not be able to generate any more queries with the same leakage anymore.*

5 Branching Garbled Circuit Chains

In this section, we construct *branching garbled circuit chains* which are the basis of our index construction. Intuitively, this new cryptographic primitive allows one to build a network of garbled circuits where each node branches into up to two other nodes and the output of the garbled circuit inside every node determines which path to take. We first recall the notion of garbled circuits.

5.1 Garbled Circuits

Garbled circuits are a well-studied cryptographic technique to "encrypt" logical circuits in a way that still preserve the functionality of the circuit but hinders the person evaluation the circuit to perform certain actions or to learn certain information. We briefly review the notation from [BHR12].

Definition 5.1 (Garbling Scheme). *A* garbling scheme *is a tuple of algorithms* $\mathcal{G} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Decode})$

$(F, e, d) \leftarrow \mathsf{Garble}(1^\lambda, f)$ *On input the security parameter λ in unary and a boolean circuit f,* Garble *outputs (F, e, d) where F is a garbled circuit, e is encoding information and d is decoding information.*

$X \leftarrow \mathsf{Encode}(e, x)$ *On input encoding information e and a input x suitable for f,* Encode *outputs a garbled input X.*

$Y \leftarrow \mathsf{Eval}(F, X)$ *On input (F, X) as above,* Eval *outputs a garbled output Y.*

$y \leftarrow \mathsf{Decode}(d, Y)$ *On input decoding information d and a garbled output Y,* Decode *outputs a plain output y.*

In most settings where garbled circuits are used, the circuit f is public. Circuit privacy can always be achieved through an universal circuit but this incurs a significant performance penalty. In our case, it is publicly known that our circuits are comparison circuits, but it should remain secret against which constant those circuits compare against. To quantify the amount of information that can be leaked without compromising security, we define the XOR-topology of a circuit.

Definition 5.2 (XOR-Topology). *The XOR-topology $\Phi_{\text{xor}}(f)$ of a circuit f is a function that maps the circuit f to a circuit where every non-XOR gate is replaced with an AND gate.*

We proceed by stating security guarantees a garbling scheme may satisfy.

Definition 5.3 (Garbled Circuit Security).

Circuit Privacy **(**$\mathsf{c\text{-}prv.sim}_\Phi$**):** *Intuitively, the collection (F, d) should not reveal any more information than $\Phi(f)$. More concretely, there must exist a simulator S that takes input $(1^\lambda, \Phi(f))$ and whose output is indistinguishable from (F, d) generated the usual way.*

Evaluation Privacy **(**$\mathsf{prv.sim}_\Phi$**):** *Intuitively, the collection (F, X, d) should not reveal any more information about x than $f(x)$. More concretely, there must exist a simulator S that takes input $(1^\lambda, \Phi(f), f(x))$ and whose output is indistinguishable from (F, X, d) generated the usual way.*

5.2 Branching Garbled Circuit Chain: Definition

Definition 5.4 (Branching Garbled Circuit Chain). *A* branching garbled circuit chain *is a tuple of algorithms* $\mathcal{G} = (\mathsf{Generate}, \mathsf{Encode}, \mathsf{Eval})$

$(F, e) \leftarrow \mathsf{Generate}(1^\lambda, f, e_0, e_1)$ *On input the security parameter λ in unary, a boolean circuit f, encoding information $\mathrm{e}_0, \mathrm{e}_1$,* Generate *outputs (F, e) where F is a branch-chained garbled circuit, and e is encoding information.*

$X \leftarrow \mathsf{Encode}(e, x)$ *On input encoding information e and a input x suitable for f,* Encode *outputs a garbled input X.*

$(b, X_b) \leftarrow \mathsf{Eval}(F, X)$ *On input (F, X) as above,* Eval *outputs a bit $b = f(x)$ and garbled inputs $X_b =$* $\mathsf{Encode}(e_{f(x)}, x)$.

We define security properties similar to garbled circuits.

Definition 5.5 (BGCC Security)**.**

Circuit Privacy ($\mathsf{c\text{-}prv.sim}_\Phi$)***:*** *Intuitively F should not reveal any more information than $\Phi(f)$. More concretely, there must exist a simulator S that takes input $(1^\lambda, \Phi(f))$ and whose output is indistinguishable from F generated the usual way.*

Evaluation Privacy ($\mathsf{prv.sim}_\Phi$)***:*** *Intuitively, the collection (F, X) should not reveal any more information about x than $(f(x), X_{f(x)})$. More concretely, there must exist a simulator S that takes input $(1^\lambda, \Phi(f), f(x), X_{f(x)})$ and whose output is indistinguishable from (F, X) generated the usual way.*

5.3 Branching Garbled Circuit Chain: Construction

Now we present our construction of a branching garbled circuit chain. It is based on an ordinary garbled circuit scheme where each garbled circuit gets augmented with a transition table. The garbled circuit also outputs a key K_0 or K_1, depending on the outcome b of the circuit evaluation. Given this key K_b and the current input labels, the evaluator should be allowed to compute the corresponding input labels for the circuit in the correct branch.

The transition table stores for each input-bit i four ciphertexts. Let $I^0[i], I^1[i]$ denote the i-th input labels for the current circuit, corresponding to the 0 and 1 value. Further, let $O_0^0[i], O_0^1[i]$ be the labels of the i-th input wire of the left child and $O_1^0[i], O_1^1[i]$ the labels of the i-th input wire of the right child. Then the i-th transfer gate stores the following values.

$$\begin{Bmatrix} E_{K_0,\, I^0[i]}(O_0^0[i]) \\ E_{K_0,\, I^1[i]}(O_0^1[i]) \end{Bmatrix}$$
$$\begin{Bmatrix} E_{K_1,\, I^0[i]}(O_1^0[i]) \\ E_{K_1,\, I^1[i]}(O_1^1[i]) \end{Bmatrix}$$

Here $E_{A,B}(X)$ denotes a double-key-cipher which we are going to implement by $E_{A,B}(X) :\overset{\text{def}}{=} \mathcal{O}(A||B) \oplus X$ with $\mathcal{O}$ a random oracle. Hence, without having both keys, it is impossible to learn any information about the plaintext. We note that there are many other instantiations of such double-key-ciphers in the literature, with different security guarantees under different assumptions but for simplicity we just resort on a random oracle in this construction. The point-and-permute technique [BHR12] is employed, which means that if the least significant bit of $I^0[i]$ is 0, the table entries are stored in the order as written and otherwise switched. This way the evaluator knows which ciphertext is the correct one without learning what bit value corresponds to the wire.

Remark 5.1. *We note, that it might be possible to reduce the size of the transfer-gate table by invoking techniques similar to those already employed in reducing gate-tables, like row reduction [NPS99] or [PSS+09], but employing those techniques is not straightforward as the labels of the child-circuit inputs can not depend on their parent circuit to maintain independence which is a useful property for being able to replenish circuits one-by-one.*

Construction 5.1. *Our branching garbled circuit chain construction is based on a linear garbling scheme. Concretely we use the half-gate garbled circuit scheme for efficiency. Let $\mathcal{G}^* = (\mathsf{Garble}^*, \mathsf{Encode}^*, \mathsf{Eval}^*, \mathsf{Decode}^*)$ be the half-gate scheme. Let f be a boolean circuit with n inputs and 1 output and $\mathcal{O}$ a random oracle.*

$\mathsf{Generate}(1^\lambda, f, e_0, e_1)$ *:*

```
1. $(F^*, e^*, d^*) \leftarrow \mathsf{Garble}^*(1^\lambda, f)$
2. Let $K_0$, respectively $K_1$ be the 0-label respectively the 1-label for the output wire in $F$.
3. $I^0 \leftarrow \mathsf{Encode}^*(e, 0^n)$
4. $I^1 \leftarrow \mathsf{Encode}^*(e, 1^n)$
5. $O_0^0 \leftarrow \mathsf{Encode}^*(e_0, 0^n)$
6. $O_0^1 \leftarrow \mathsf{Encode}^*(e_0, 1^n)$
7. $O_1^0 \leftarrow \mathsf{Encode}^*(e_1, 0^n)$
8. $O_1^1 \leftarrow \mathsf{Encode}^*(e_1, 1^n)$
9. for $i = 1, \ldots, n$ do:
   (a) $b' = \mathrm{lsb}(I^0[i])$
   (b) $T_d^0[i] \leftarrow \mathcal{O}(K_d \,\|\, I_{b'}[i]) \oplus O_d^{b'}[i] \quad \forall d \in \{0,1\}$
   (c) $T_d^1[i] \leftarrow \mathcal{O}(K_d \,\|\, I_{1-b'}[i]) \oplus O_d^{1-b'}[i] \quad \forall d \in \{0,1\}$
10. Let $F = (F^*, d^*, T_d^b[i])$, $e = e^*$ and output $(F, e)$
```

$\mathsf{Encode}(e, x)$ *is the same as* $\mathsf{Encode}^*(e, x)$

$\mathsf{Eval}(F, X)$ *:*

```
1. Parse the input as $F = (F^*, d^*, T_d^b[i])$.
2. $b \leftarrow \mathsf{Decode}^*(d^*, \mathsf{Eval}^*(F^*, X))$
3. Let $K_b$, be the $b$-label of the output wire
4. for $i = 1, \ldots, n$ do:
   (a) $b' = \mathrm{lsb}(X[i])$
   (b) $X_b[i] \leftarrow T_d^{b'}[i] \oplus \mathcal{O}(K_b \,\|\, X[i])$
5. output $(b, X_b)$
```

Theorem 5.2. *Construction 5.1 satisfies the BGCC definition.*

Proof. The correctness instantly follows from the correctness of the garbling scheme. □

Theorem 5.3. *Construction 5.1 satisfies the BGCC security definitions* $\mathsf{c\text{-}prv.sim}_{\Phi_{\mathrm{xor}}}$ *and* $\mathsf{prv.sim}_{\Phi_{\mathrm{xor}}}$.

Proof. Security in the random oracle model follows because the half gate garbling scheme satisfies $\mathsf{c\text{-}prv.sim}_{\Phi_{\mathrm{xor}}}$ and $\mathsf{prv.sim}_{\Phi_{\mathrm{xor}}}$ (see Lemma A.2, A.3) and furthermore the random oracle is with probability $1 - \mathrm{negl}(\lambda)$ never evaluated more than once on the same input, hence the transition table is indistinguishable from random. □

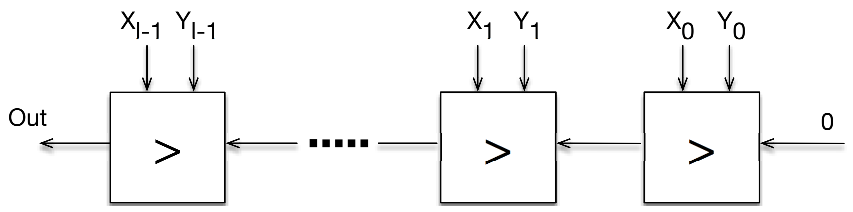


Figure 1: Schematic diagram of a garbled comparison circuit

5.4 Half-Gates Garbling Scheme

We use the half-gate garbling scheme [ZRE15] because it is the fastest garbling scheme known, both in garbling and evaluation time and, more importantly, size of the garbled circuit.

It has the following properties:

1. It is compatible with Free-XOR [KS08] i.e. garbling and evaluating XOR gates does only involve cheap XOR operations and XOR gates disappear in the garbled circuit representation.
2. It supports XOR gates and gate with an odd number of true-values (e.g. AND, NAND, . . .).
3. Other gates, like $f(x, y) = 1$ or $f(x, y) = x$ are not supported.
4. Security holds if the used hash function is "circular correlation robust". The used construction of the hash function is circular correlation robust in the random permutation model. The random permutation is instantiated with AES.
5. The scheme satisfies the definition $\mathsf{prv.sim}_{\Phi_{\text{xor}}}$ and $\mathsf{c\text{-}prv.sim}_{\Phi_{\text{xor}}}$ (c. f. Lemma A.3, A.2).

5.5 Comparison Circuits

The circuits we are interested in in our scheme are comparison circuits. There are many proposed ways to implement such a circuit in the literature but in our scenario we have the very different goal of minimizing the non-XOR gates in the circuit. Physical layout, latency, heat, etc. are irrelevant for us.

5.5.1 Generic Comparison Circuit

An ℓ-bit comparison circuit takes as input two ℓ-bit integers $a = (a_{\ell-1}, \ldots, a_0), b = (b_{\ell-1}, \ldots, b_0)$ represented in binary with the most significant bit first and outputs $(a < b)$, i.e. 1 if $(a < b)$ and 0 if $(a \geq b)$. We use the comparison circuit of [KSS09] that, for comparing two ℓ-bit integers needs only ℓ non-XOR gates, which is optimal. The ℓ-bit comparison circuit is a concatenation of ℓ one-bit comparators. A one-bit comparator takes as input three bits: one bit of a, one bit of b and a carry bit c. The output z of the one-bit comparator is defined as follows:

$$z = \begin{cases} c & a = b \\ 1 & a < b \\ 0 & a > b \end{cases}$$

The correct one-bit comparator circuit is implemented as $z = b \oplus ((a \oplus c) \wedge (b \oplus c))$. The ℓ-bit comparator is just the concatenation of ℓ one-bit comparators with the initial carry $c_0 = 0$. The invariant that holds in each step is that when the i-th one-bit comparator is processed, the carry $c_{i+1} = ((a_i, \ldots, a_0) < (b_i, \ldots, b_0))$. Hence per induction the final output $c_{\ell+1}$ is correct.

5.5.2 Comparison Against a Constant

The circuits we are really interested in in our scheme are not generic comparison circuits but in fact circuits that compare the input against a hardwired constant b. To achieve this, we could garble a generic comparison circuit and store the correct input labels for the bits of b along with it. But we do not want to store the additional ℓ ciphertexts for the b-labels as this would be a significant overhead. Instead we directly want to garble a circuit that already represents the desired functionality. In terms of the comparison circuit, the values of b are known in advance and thus the one-bit comparator reduces to a single AND gate if $b_i = 0$ and a single NOR gate, if $b = 1$. Altogether we need to store two ciphertexts for each one-bit comparator.

5.6 Performance Considerations

Some readers may have the preconception that garbled circuits incur significant performance overheads in general. This is not true in our setting. A garbled 32-bit comparison circuit is only 1040 byte in our implementation and evaluating it takes only 64 AES evaluations of which 32 come for free as they are independent and hence can exploit instruction level parallelism. A single AES instruction has a latency of 7 cycles on modern CPUs, hence the complete evaluation of the circuit can theoretically be as fast as 224 cycles.

6 Our Reactive Index Construction

Before presenting our scheme, we present a strawman interactive scheme.

6.1 Interactive Reactive Index

Consider the key-value pairs arranged in a binary search tree, sorted by key, where each node is IND-CPA encrypted with a master secret key. To search the tree for a particular value a, the server needs a mechanism to compare an encryption of a against the encrypted data within each node and decide whether to go left or right in the tree. Since the data is encrypted with an IND-CPA encryption scheme, the server cannot perform this comparison on encrypted data. A solution is to fetch the encrypted value from the server, decrypt it, compare it against the plaintext and send the comparison result to the server, as is currently done in mOPE [PLZ13] and ZeroDB [EW16]. The downside of this solution is the large network latency introduced by multiple network roundtrips between the client and the server's data center, resulting in significant overhead. Our scheme, which is non-interactive, provides an order of magnitude increased performance, as discussed in §10.

6.2 Improving Security

The above scheme is not only inefficient, but it provides weaker security than desired. The reason is that the shape of the tree (to some degree) depends on the order of the operations performed on the tree. When an attacker steals a database, the attacker can learn some information about the data and the operations from the shape of the tree, despite the nodes being encrypted with IND-CPA security. To fix this problem, we use a treap data structure instead of a regular balanced search tree.

A treap, also sometimes known as a randomized binary search tree is a probabilistic tree data structure with *expected* runtime guarantees. Its name is a portmanteau of tree and heap. It is a Cartesian tree in which each key is given a (randomly chosen) numeric priority. As with any binary search tree, the inorder traversal order of the nodes is the same as the sorted order of the keys. The structure of the tree is determined by the requirement that it be heap-ordered: that is, the priority number for any non-leaf node must be greater than or equal to the priority of its children. Thus, as with Cartesian trees more generally, the root node is the maximum-priority node, and its left and right subtrees are formed in the same manner from the subsequences of the sorted order to the left and right of that node.

We now go through some interesting properties of this datastructure which we make use of in our construction.

6.2.1 Treap Properties

Remark 6.1. *In the description of treaps it is usually assumed that inserted keys are unique. We extend this by allowing duplicates and resolve conflicts by using the "<" operator for comparison during inserts (i.e. duplicate keys go to the right).*

Definition 6.1 (Shape of a Binary Search Tree)**.** *We define the* shape *of a binary search tree to be the underlying directed acyclic graph with edges being either marked as left or right, corresponding to the original tree.*

Definition 6.2 (Rank of an Insert)**.** *For a sequence of insert queries $Q = \{q_1, \dots, q_n\}$ with q_i represented as a pair of a key and the number in the sequence, i.e. $q_i = (v_i, i)$, define the rank of a query as $rank(q_i) :\overset{\text{def}}{=} |\{(v', i') \in Q | v' < v_i \vee (v' = v \wedge i' < i)\}|$ the number of queries in the sequence that either insert a smaller key or the same key and come before.*

Fact 6.2. *If a sequence of insert queries $Q = \{q_1, \dots, q_n\}$ generates a treap with certain shape, then the sequence of inserts $Q' = \{(\text{rank}(q_1), 1), \dots, (\text{rank}(q_n), n)\}$ generates a treap with the same shape.*

Fact 6.3. *For a sequence of insert queries $Q = \{q_1, \dots, q_n\}$ where each insert query is already assigned the random treap-priority, the order in which the queries are executed does not influence the shape of the final treap. The final treap will always have the same shape as the binary search tree that would be generated if the queries were executed in the order of their priority without rebalancing.*

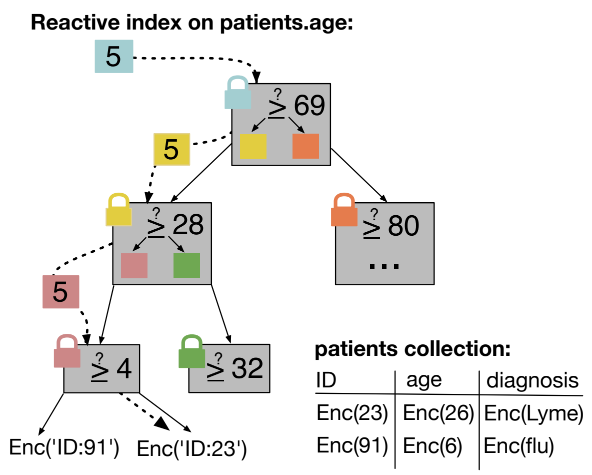
6.3 A Secure Non-Interactive Index

We now explain our non-interactive scheme. The server maintains a treap similar to the one in the interactive scheme but instead of storing the IND-CPA secure encryption of the key-value pair at each node, the treap forms a branching garbled circuit chain with each node corresponding to a garbled comparison circuit, that compares its input against the key associated with the node.

Since garbled circuits may only be used once, the client will have to provide new such garbled circuits to replace the used ones. We call this the *repair* or *replenishment* procedure. Fortunately, the expected number of garbled circuits consumed is logarithmic and furthermore they can be replenished without consuming the other nodes in the tree.

6.4 GC Replenishing

To repair an index that contains consumed garbled circuits the server sends the following request to the client:



ID	age	diagnosis
Enc(23)	Enc(26)	Enc(Lyme)
Enc(91)	Enc(6)	Enc(flu)

Figure 2: Overview of our reactive index.

- All unique garbled circuit ids of the immediate children of the consumed nodes, but not the ones of the consumed nodes themselves.
- The encrypted keys (note, these are the keys as in “key-value” pairs, not cryptographic keys) of the consumed nodes.
- Information about where these nodes are located in the tree.

In the execution of our protocol, this data always corresponds to a path in the index. We start with the leaf and generate a new garbled circuit with a new random id. For the parent we also create a new garbled circuit with a new random id. Additionally we have to provide a transition table for this circuit. Therefore we need the input labels of the child circuits. We have the labels of one child still in memory and the labels of the other child can be generated by their corresponding circuit’s unique id which was part of the replenishing request.

6.5 Inserts and Deletes: Rebalancing

The insertion or deletion of a key-value pair may require rotations to preserve the heap properties. In any case, only nodes that were already consumed for finding the node are rotated so the rotations have no effect on which nodes need to be replenished.

6.6 Queries With Two Bounds

To answer queries with two bounds in one round, we store two copies of the index at the server, where the first index is used to locate the first bound and the second is used to locate the second bound. It does not suffice to only store one copy of the index because the search for the first bound can damage nodes that are on the path to the second bound.

6.7 Continuous Aggregation

Our index can also be used to aggregate values within a certain range and compute a function such as SUM, AVERAGE, or PRODUCT very efficiently. Previous constructions relied on partially homomorphic encryption schemes which are orders of magnitude slower than AES. Our reactive index is solely based on fast symmetric cryptography. To be able to compute the aggregate over any continuous range, each node in the index additionally stores an encrypted aggregate of the values of all children. To determine the aggregate over a continuous range of values, the server computes a perfect cover of the range, which can be computed by traversing the tree in the same manner as the previously described RANGE scheme. The required covering set is $\mathcal{O}(\log n)$ and it is sent to the client, who decrypts the partial aggregates and aggregates the plaintexts to get the final result.

7 One-Round Mutable Order Preserving Encryption

A side product of our range-query scheme is one-round order preserving encryption. It can be achieved by using the mOPE construction [PLZ13] and applying the branched garbled circuit chain to remove interactions.

8 Security Guarantees and Proofs

In this section we formally define the security guarantees our reactive index construction provides for each query. Specifically this means to define a leakage function Leak for each query and prove our security definition with this leakage function.

Definition 8.1 (Rank of an Element)**.** *We define the Rank of an Element $rank(x)$ in a list $L = (a_i)_{i\in\mathbb{N}}$ as* $\mathrm{rank}(L, x) :\overset{\text{def}}{=} |\{a_i \mid a_i \leq x\}|$ *and we write* $\mathrm{rank}(x)$ *if L is clear from the context.*

For range queries, the attacker learns the rank of the lower and upper bound of the range boundaries. The same is the case for aggregation queries. For order-by-limit queries, the attacker only learns the rank of the lower bound. In the case of inserts/deletes, the attacker will learn the rank of the respective element, more precisely:

Theorem 8.1. *The described reactive index is non-adaptively secure in the sense of indistinguishability for all supported queries from definition 4.3 with the following leakage functions:*

$$
\begin{aligned}
\mathsf{Leak}(DB) &= |DB| \\
\mathsf{Leak}(DB, (\mathit{RANGE}, [a, b])) &= (\mathit{RANGE}, \mathrm{rank}(a-1), \mathrm{rank}(b)) \\
\mathsf{Leak}(DB, (\mathit{ORDER\ \ LIMIT}, [a, b], \mathrm{start}, \mathrm{length})) &= (\mathit{ORDER\ \ LIMIT}, \mathrm{rank}(a-1), \mathrm{rank}(b), \mathit{start}, \mathit{length}) \\
\mathsf{Leak}(DB, (\mathit{AGGREGATE}, [a, b])) &= (\mathit{AGGREGATE}, \mathrm{rank}(a-1), \mathit{rank}(b)) \\
\mathsf{Leak}(DB, (\mathit{INSERT}, k, v)) &= (\mathit{INSERT}, \mathrm{rank}(k)) \\
\mathsf{Leak}(DB, (\mathit{DELETE}, [a, b]])) &= (\mathit{DELETE}, \mathrm{rank}(a-1), \mathrm{rank}(b))
\end{aligned}
$$

Proof. We divide the proof into two parts. First, we show that the initial indices after the execution of the first phase, where the leakage is only the size of the database, are indistinguishable. Then we continue with proving that each query is indistinguishable.

Indistinguishability of the Initial Indices. Let $\mathcal{I}_m^0$ respectively $\mathcal{I}_m^1$ be the indices generated after the first m queries are applied. We now prove $\mathcal{I}_m^0 \overset{c}{\approx} \mathcal{I}_m^1$.

First, the shape of both indices is perfectly indistinguishable through the treap properties. Specifically, the facts 6.2 and 6.3 say that the set of inserts $Q' = \{rank(q_1), \ldots, rank(q_n)\}$, which is exactly $\{1, \ldots, |\mathcal{DB}|\}$, generates the same shape as any other set of inserts of size $|\mathcal{DB}|$. Without loss of generality we therefore from now on assume the shapes of $\mathcal{I}_m^0$ and $\mathcal{I}_m^1$ were the same.

We proceed by showing that a node $v \in \mathcal{I}_m^0$ and the corresponding node $v' \in \mathcal{I}_m^1$ are computationally indistinguishable: $v \overset{c}{\approx} v'$. Each node contains a BGCC, an encryption of the key and an encryption of the payload (the value). The BGCCs are indistinguishable (Theorem 5.3) and the latter two are indistinguishable through the properties of the encryption scheme.

Indistinguishability of the Queries. Given, that each pair of queries produces the same leakage, the outputs of the BGCCs in the execution are the same. If follows with theorem 5.3 that the encrypted queries are indistinguishable.

Finally, the indistinguishability of the repair data $\mathcal{H}$ follows with the same arguments as in the indistinguishability of the initial indices.

□

We remark that non-adaptive security is all that is needed in our threat model because the attacker does not have control over the queries and the client sends queries without seeing the encrypted data. An adaptive security proof is also possible with tweaks to the protocol that complicate it.

Corollary 8.2. *The described reactive index satisfies the definition of non-adaptive semantic security.*

Proof. With the preceding observation in theorem 4.3, it only remains to show that our leakage function Leak is reproducible. We claim that the following method, given leakage

$$x = (x_0, \ldots, x_n) = (\mathsf{Leak}(\mathcal{DB}^*), \mathsf{Leak}(q_1^*), \ldots, \mathsf{Leak}(q_n^*))$$

produces a database $\mathcal{DB}$ and queries $(q_1, \ldots, q_n)$ that produce the same leakage. We start with an initial database $\mathcal{DB} = ((1, \perp), \ldots, (x_0, \perp))$ and gradually adjust it based on the queries. For bookkeeping we maintain a list $\mathcal{D}$ of deleted elements. For $i = 1, \ldots, n$, do:

if $x_i = (\texttt{RANGE}, r_0, r_1)$
    set $q_i = (\texttt{RANGE}, r_0 + \text{rank}(\mathcal{D}, r_0), r_1 + \text{rank}(\mathcal{D}, r_1))$

if $x_i = (\texttt{ORDER LIMIT}, r_0, r_1, s, l)$
    set $q_i = (\texttt{ORDER LIMIT}, r_0 + \text{rank}(\mathcal{D}, r_0), r_1 + \text{rank}(\mathcal{D}, r_1), s, l)$

if $x_i = (\texttt{AGGREGATE}, r_0, r_1)$
    set $q_i = (\texttt{AGGREGATE}, r_0 + \text{rank}(\mathcal{D}, r_0), r_1 + \text{rank}(\mathcal{D}, r_1))$

if $x_i = (\texttt{INSERT}, k)$
    We make room in $\mathcal{DB}$ for an insert at position $k + \text{rank}(\mathcal{D}, k)$ and adjust our previous choice for the queries $q_1, \ldots, q_i$ accordingly. Concretely, we add $(|\mathcal{DB}| + 1, \perp)$ to $\mathcal{DB}$ and in the queries $q_1, \ldots, q_i$ we increment all tokens that are greater or equal to $k + \text{rank}(\mathcal{D}, k)$ by one. Then we set $q_i = (\texttt{INSERT}, k + \text{rank}(\mathcal{D}, k), \perp)$

if $x_i = (\texttt{DELETE}, r_0, r_1)$
    set $q_i = (\texttt{DELETE}, [r_0 + \text{rank}(\mathcal{D}, r_0), r_1 + \text{rank}(\mathcal{D}, r_1)])$ and
    set $\mathcal{D} :\overset{\text{def}}{=} \mathcal{D} \cup (\mathcal{DB} \cap [r_0, r_1])$

At the end we have

$$(\mathsf{Leak}(\mathcal{DB}), \mathsf{Leak}(q_1), \ldots, \mathsf{Leak}(q_n)) = (\mathsf{Leak}(\mathcal{DB}^*), \mathsf{Leak}(q_1^*), \ldots, \mathsf{Leak}(q_n^*))$$

as desired.

□

8.1 Ideal Security Guarantees for k Query Observations

Under the assumption that the adversary observes at most k queries, where k is a small constant that is fixed in advance, we can get better security guarantees with a slight modification of the scheme. This for example models the case of a database server that supports the concurrent execution of k queries and an attacker that obtains a copy of the server disk and memory at one single point in time. Concretely, we can achieve to only leak the size of the result set (which is ideal) as opposed to leaking the size of the result set plus the size of the two sets that are below, respectively above, the range.

We achieve this by having k distinct indices, and *permuting* each of them by a random, secret offset. The circuits need to be adjusted accordingly to be aware of this offset.

9 Implementation

9.1 libgarble

We implemented a library for garbling circuits in C++ which we call libgarble [libgarble]. The goals of the library are similar to JustGarble [BHK+13],[JustGarble]: It provides plain garbling of circuits, efficient import and export of garbled circuits and circuit layouts and nothing else. In the future there will also functions to convert circuits from the abundant other file formats into the libgarble file format. libgarble improves on JustGarble in that it implements the recent half-gate optimization, fixes some security problems, has options to keep the circuit layout separate form the garbled circuit gate tables, and is more portable across different platforms, specifically it can also run on platforms where AES-NI is not available. Also, we provide a JNI interface through which we can call libgarble from Java applications. In our tests, the JNI interface does not add noticeable overhead compared to calling libgarble directly from C. libgarble will be published under a free software license and we will maintain an open and active development with the goal to make the library ready for production use.

9.2 Secure Reactive Index

We implement a prototype of our secure reactive index construction in Java with the goal to evaluate the feasibility of our approach. The implementation runs stable with data sets of 1 million elements but is not ready for production use, yet. We also plan to open-source the reactive index as soon as it has reached a sufficiently maturity.

9.3 Limitations

As Java is a high-level programming language with garbage collection, our implementation can not achieve the exact security guarantees as in our definition as we would need to be able to securely erase data and also be able to precisely control where in memory our objects reside. Even if the logical structure inside our Java code does not leak information about the history of inserts, the Java virtual machine might not have securely deleted data we deleted and it may also allocate and manage memory in a way that leaks the history of execution to an attacker. A secure implementation needs to be done in a low-level language like C++ that gives more control over memory management. But this does not affect the validity of our evaluation results.

10 Evaluation

The goal of this research was to find a solution that has comparable performance to a system that performs the operations on plaintext data. The predominant factor is the network latency of roughly 30ms round-trip time across North America or even 70ms Trans Atlantic. The overhead cost of our construction should not add significantly to this number. There are two main concerns that could render our construction slow. First is the size of the garbled circuits what we need to store for each entry in the index. Second is the time to evaluate garbled circuits. Our evaluation shows that both factors are moderate compared to network latency.

One garbled circuit has the size of two ciphertexts per gate and additionally needs to store a 128 bit unique random id. An n bit const-comparison circuit has exactly n gates. No additional metadata is needed, hence the size for a n bit comparison circuit is $n \cdot 2 \cdot 128 + 128$ bit. For a 32 bit comparator this amounts to 1040 byte. One transition table has the size $n \cdot 2 \cdot 128$ bit. Hence the size of a complete node is $n \cdot 6 \cdot 128 + 128$ bit, which again for a bitlength of 32 bit results in 3088 byte.

The cost of evaluating a 32 bit comparison circuit is dominated by the 64 AES evaluations needed. Theoretically of these 64 evaluations, 32 come for free as they are independent and hence can exploit instruction level parallelism. A single AES instruction has a latency of 7 cycles on modern CPUs, hence the complete evaluation of the circuit can theoretically be as fast as 224 cycles, at least for the AES part. For a 32-bit constant comparison circuit, in our current implementation the time to garble is 19786 cycles and the time to evaluate is 7842 cycles. For a 128-gate circuit it is 70109/29099 cycles. The overhead stems from the fact that we currently use the gcrypt library to evaluate AES and we have not yet done any low-level optimizations.

For a complete system evaluation, we bulk-loaded 1 million random 32 bit integers and ran two iterations of 1000 random inserts, deletes and range-queries each, where the range-query ranges were increased gradually from 1 to 1000. Bulk-loading 1M elements took 63 seconds. On average, an INSERT took 4.3ms, a RANGE query 6.8ms, and a DELETE 4.6ms.

11 Future Work

Further performane improvements. Our reactive index is integrated in the new Arx database system. We are currently in the process of optimizing the performance of our implementation, especially for scalability and concurrency.

Protection Against a Malicious Attacker. To protect against a malicious adversary at the server, one can extend our reactive index construction with Merkle trees. The client computes a Merkle tree over the entire treap, including over the garbled circuits and stores the root hash. When the server provides tree leaves corresponding to the order queries requested, the server attaches a proof of correctness of these leaves by incorporating the relevant paths of hashes. Furthermore, as the client repairs the index, the client can recalculate incrementally the root hash. The details remain to be worked out and it also remains to be seen how this extension affects performance.

Hiding Access Patterns. In the current security definition (and construction), the server may see the access patterns consisting of paths down the tree (effectively corresponding to the index of the bounds). To hide these patterns, one can use an oblivious RAM data structure. In particular, instead of using ORAM as a black-box, an oblivious tree data structure is more efficient [WNL+14]. The details remain to be worked out and it remains to see how efficient this construction would be, although likely the slowdown will not justify the slightly increased security in many applications.

12 Conclusion

We constructed and evaluated the first practical, functionally rich index that allows efficient computation of standard database queries on encrypted data and we proved our scheme secure in a novel, precisely defined, and well motivated, security model.

13 Acknowledgements

We thank Riyaz Faizullabhoy for helpful discussions.

Appendices

A More Details on the Half Gate Scheme

The half gate scheme uses a hash function H that needs to satisfy correlation robustness for naturally derived keys (definition A.1). This can be achieved in the random permutation model with the following construction.

Construction A.1. *For a random permutation $\pi : \{0,1\}^k \to \{0,1\}^k$, we define the hash function $H_\pi(x,i)$ to be $\pi(2x \oplus i) \oplus 2x \oplus i$*

Definition A.1. *Given a hash function H, we define two oracles:*

- $\mathrm{Circ}_{\mathcal{R}}(x,i,b) : \overset{\mathrm{def}}{=} H(x \oplus R, i) \oplus bR$, *where* $R \in \{0,1\}^{k-1}1$
- $\mathrm{Rand}(x,i,b)$ *is a random function with k-bit output.*

Say that a sequence of queries of the form (x,i,b) to an oracle $\mathcal{O}$ are natural if they satisfy the following:

- *for the q-th query, we have $i = q$.*
- $b \in \{0,1\}$
- *x is naturally derived, meaning that it is obtained from one of these operations:*
- $x \leftarrow \{0,1\}^k$
- *$x \leftarrow x_1 \oplus x_2$, where x_1 and x_2 are naturally derived*
- *$x \leftarrow H(x_1,i)$, where x_1 is naturally derived and $i \in \mathbb{Z}$*
- *$x \leftarrow \mathcal{O}(x_1,i,b)$, where x_1 is naturally derived*

Algorithm 4: Garbling one gate in the half-gate scheme

```
input  : The incoming wire labels $W_a^0, W_a^1, W_b^0, W_b^1$ for the wires $a, b$, the global offset $R$, and the gate_type
output : wire label $W_{\text{out}}^0$
1  $p_a = \texttt{lsb}(W_a^0)$
2  $p_b = \texttt{lsb}(W_b^0)$
   /* fun is a function that outputs two bits based on a gate_type */
3  $(\alpha_a, \alpha_b) = \texttt{fun}(\text{gate\_type})$
4  $G = H(W_a^0) \oplus H(W_a^1)$
5  if $p_b \neq \alpha_b$ then
6      $G = G \oplus R$
7  if $p_a = 0$ then
8      $W_G^0 = H(W_a^0)$
9  else
10     $W_G^0 = H(W_a^1)$
11 if gate_type $(p_a, p_b) = 1$ then
12     $W_G^0 = W_G^0 \oplus R$
13 $E = H(W_b^0) \oplus H(W_b^1)$
14 if $\alpha_a = 0$ then
15     $E = E \oplus W_a^0$
16 else
17     $E = E \oplus W_a^1$
18 if $p_b = 0$ then
19     $W_E^0 = H(W_b^0)$
20 else
21     $W_E^0 = H(W_b^1)$
22 $W_{\text{out}^0} = W_G^0 \oplus W_E^0$
```

Then H is circular correlation robust for natural keys *if, for all polynomial-time adversaries* Adv *making natural queries,*

$$\left|\Pr[\mathsf{Adv}^{\mathrm{Circ}_{\mathcal{R}}}(1^k) = 1] - \Pr[\mathsf{Adv}^{\mathrm{Rand}}(1^k) = 1]\right| \leq \mathrm{negl}(k)$$

Fact A.1. *In the half gate garbling scheme, all wire labels are naturally derived and H from construction A.1 is correlation robust for naturally derived keys.*

Lemma A.2. *The half-gate garbling scheme satisfies* $\mathsf{prv.sim}_{\Phi_{\text{xor}}}$

Proof. The proof is exactly the same as the proof of theorem 1 in [ZRE15], which states that the scheme satisfies $\mathsf{prv.sim}_f$ with the whole circuit f as leakage. In their proof the simulator only makes its decisions based on whether a particular gate is an XOR gate or not and does not depend on the concrete type of the gate if it is not an XOR gate. Hence the stronger property $\mathsf{prv.sim}_{\Phi_{\text{xor}}}$ is also satisfied. □

Lemma A.3. *The half-gate garbling scheme satisfies* $\mathsf{c\text{-}prv.sim}_{\Phi_{\text{xor}}}$

Proof. To proof this theorem we need to have a closer look at the process for garbling one gate, as written down in algorithm 4. We use the same notation as in [ZRE15].

The entire process does not reveal the type of the gate because W_a^0, W_b^0 are naturally derived and hence indistinguishable from random. Therefore p_a, p_b are indistinguishable from random as well. The same holds true for W_G^0 and R is chosen completely random. Hence, looking at what the algorithm does, it follows that different gate types are indistinguishable.

$\square$

References

[JustGarble] *JustGarble*. URL: http://cseweb.ucsd.edu/groups/justgarble/.

[libgarble] *libgarble*. URL: http://libgarble.org/.

[TPC-C] *TPC-C*. http://www.tpc.org/tpcc/.

[ABS+15] Prabhanjan Ananth, Zvika Brakerski, Gil Segev, and Vinod Vaikuntanathan. “From Selective to Adaptive Security in Functional Encryption”. In: *Proceedings of the 35th International Cryptology Conference (CRYPTO)*. Santa Barbara, CA, Aug. 2015.

[AEE+09] Divyakant Agrawal, Amr El Abbadi, Faith Emekci, and Ahmed Metwally. “Database Management as a Service: Challenges and Opportunities”. In: *Proceedings of the 25th International Conference on Data Engineering (ICDE)*. 2009.

[AKS+04] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. “Order preserving encryption for numeric data”. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. Paris, France, June 2004.

[AWW12] George Weilun Ang, John Harold Woelfel, and Terrence Peter Woloszyn. *System and Method of Sort-Order Preserving Tokenization*. US Patent Application 13/450,809. 2012.

[BCL+09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O’Neill. “Order-Preserving Symmetric Encryption”. In: *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*. Cologne, Germany, Apr. 2009, pp. 224–241.

[BCO11] Alexandra Boldyreva, Nathan Chenette, and Adam O’Neill. “Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions”. In: *Advances in Cryptology (CRYPTO)*. Aug. 2011, pp. 578–595.

[BHK+13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Philip Rogaway. “Efficient Garbling from a Fixed-Key Blockcipher”. In: *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*. San Francisco, CA, May 2013.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. “Foundations of Garbled Circuits”. In: *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. Raleigh, NC, Oct. 2012, pp. 784–796.

[BLR+14] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. “Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation”. In: *Proceedings of the 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*. 2014.

[BW07] Dan Boneh and Brent Waters. “Conjunctive, subset, and range queries on encrypted data”. In: *Proceedings of the 4th Theory of Cryptography Conference (TCC)*. 2007.

[CGP+15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. “Leakage-Abuse Attacks against Searchable Encryption”. In: *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, CO, Oct. 2015.

[Cip] CipherCloud. *Tokenization for Cloud Data*. http://www.ciphercloud.com/tokenization-cloud-data.aspx.

[CLW+16] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. "Practical Order-Revealing Encryption with Limited Leakage". In: *Proceedings of the 23rd International Conference on Fast Software Encryption (IACR-FSE)*. 2016.

[EW16] Michael Egorov and MacLane Wilkison. "ZeroDB white paper". In: *CoRR* abs/1602.07168 (2016). URL: http://arxiv.org/abs/1602.07168.

[FJK+15] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. "Rich Queries on Encrypted Data: Beyond Exact Matches". In: *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS)*. Vienna, Austria, Sept. 2015.

[GGH+16] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. "Functional Encryption Without Obfuscation". In: *Proceedings of the 13th Theory of Cryptography Conference (TCC)*. 2016.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. "Black-Box Garbled RAM". In: *Proceedings of the 56th Annual Symposium on Foundations of Computer Science (FOCS)*. 2015.

[GMP15] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. *TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption*. Cryptology ePrint Archive, Report 2015/1010. http://eprint.iacr.org/. 2015.

[Goh03] Eu-Jin Goh. *Secure Indexes*. Cryptology ePrint Archive, Report 2003/216. http://eprint.iacr.org/. 2003.

[HIL+02] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. "Executing SQL over encrypted data in the database-service-provider model". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. Madison, WI, June 2002.

[KAK10] Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa. "MV-OPES: Multivalued-Order Preserving Encryption Scheme: A Novel Scheme for Encrypting Integer Value to Many Different Values". In: *IEICE Trans. on Info. and Systems* E93.D.9 (2010).

[Ker15] Florian Kerschbaum. "Frequency-Hiding Order-Preserving Encryption". In: *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, CO, Oct. 2015, pp. 656–667.

[KS08] Vladimir Kolesnikov and Thomas Schneider. "Improved Garbled Circuit: Free XOR Gates and Applications". In: *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*. 2008, pp. 486–498.

[KS14] Florian Kerschbaum and Axel Schröpfer. "Optimal Average-Complexity Ideal-Security Order-Preserving Encryption". In: *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. Scottsdale, AZ, Nov. 2014, pp. 275–286.

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. "Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima". In: *Proceedings of the 8th International Conference on Cryptology and Network Security (CANS)*. 2009, pp. 1–20.

[LPL+09] Seungmin Lee, Tae-Jun Park, Donghyeok Lee, Taekyong Nam, and Sehun Kim. "Chaotic Order Preserving Encryption for Efficient and Secure Queries on Databases". In: *IEICE Trans. on Info. and Systems* E92.D.11 (2009).

[Lu12] Yanbin Lu. “Privacy-Preserving Logarithmic-time Search on Encrypted Data in Cloud”. In: *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2012.

[LW12] Dongxi Liu and Shenlu Wang. “Programmable Order-Preserving Secure Index for Encrypted Database Query”. In: *IEEE International Conference on Cloud Computing*. 2012.

[LW13] Dongxi Liu and Shenlu Wang. “Nonlinear order preserving index for encrypted database query in service cloud environments”. In: *Concurrency and Computation: Practice and Experience* (2013).

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. “Privacy Preserving Auctions and Mechanism Design”. In: *Proceedings of the 1st International Conference on Cryptology and Network Security (CANS)*. 1999, pp. 129–139.

[ÖSC03] Gultekin Özsoyoglu, David A. Singer, and Sun S. Chung. “Anti-Tamper Databases: Querying Encrypted Databases”. In: *IFIP WG 11.3 Working Conf. on Database and Applications Security*. 2003.

[Per] Perspecsys. *The PRS Server: Data Protection for Cloud Applications*. http://www.perspecsys.com/perspecsys-cloud-protection-gateway/.

[PLZ13] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. “An Ideal-Security Protocol for Order-Preserving Encoding”. In: *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*. San Francisco, CA, May 2013.

[PSS+09] Benny Pinkas, Thomas Schneider, Nigel Smart, and Stephen Williams. “Secure Two-Party Computation Is Practical”. In: *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Tokyo, Japan, 2009, pp. 250–267.

[Ras11] Fahmida Y. Rashid. “Salesforce.com Acquires SaaS Encryption Provider Navajo Systems”. In: *eWeek.com* (2011).

[SBC+07] Elaine Shi, John Bethencourt, T-H. Hubert Chan, Dawn Song, and Adrian Perrig. “Multi-dimension range query over encrypted data”. In: *Proceedings of the 28th IEEE Symposium on Security and Privacy (IEEE S&P)*. Oakland, CA, May 2007.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Song. “Towards Practical Oblivious RAM”. In: *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2012.

[SW15] Muhammad Naveed and Seny Kamara and Chares V. Wright. “Inference Attacks on Property-Preserving Encrypted Databases”. In: *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, CO, Oct. 2015.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. “Oblivious Data Structures”. In: *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. Scottsdale, AZ, Nov. 2014.

[XYH12] Liangliang Xiao, I-Ling Yen, and Dung T. Huynh. *Extending Order Preserving Encryption for Multi-User Systems*. Cryptology ePrint Archive, Report 2012/192. 2012.

[YKK+11] Dae Yum, Duk Kim, Jin Kim, Pil Lee, and Sung Hong. “Order-Preserving Encryption for Non-uniformly Distributed Plaintexts”. In: *Intl. Workshop on Information Security Applications*. 2011.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. “Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates”. In: *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*. 2015.