Future Generation Computer Systems 36 (2014) 441–452

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Privacy/performance trade-off in private search on bio-medical data

H. Perl [a,*], Y. Mohammed [b], M. Brenner [a], M. Smith [a]

[a] Distributed Computing and Security Group, Leibniz University Hannover, Schloßwender Straße 5, 30159 Hannover, Germany

[b] Biomolecular Mass Spectrometry Unit, Leiden University Medical Center, Einthovenweg 20, 2333 ZC Leiden, The Netherlands

HIGHLIGHTS

- We improve and elaborate on a new fast and secure exact term search algorithm with private queries.
- The algorithm offers a trade-off between privacy and performance.
- The results can then be further aggregated using Homomorphic Cryptography.
- We expose these functionalities as a ready-to-use Web service.

ARTICLE INFO

Article history:
Received 1 February 2013
Received in revised form
4 November 2013
Accepted 3 December 2013
Available online 16 December 2013

Keywords:
Privacy
Homomorphic Cryptography
Bloom filters
Secure search

ABSTRACT

Outsourcing of biomedical data, especially human patient data, for processing is heavily constrained by legal issues. For instance searching for a biological sequence of amino acids or DNA nucleotides in a library or database of sequences of interest to identify similarities is not something which can easily be outsourced due to the data protection and privacy laws. However, DNA sequencing is becoming a main stream technology, thus it would be desirable to be able to offer computational services without endangering the patient privacy. While data in transit can easily be protected by transport layer security, the data must be stored in the clear during processing. Most algorithms and schemes are either optimized for speed with no consideration for data protection and thus cannot be used to offer services. On the other hand the theoretical Private Information Retrieval (PIR) schemes that protect the privacy of patient data are so slow that they are not feasible for the real world use. Since the search spaces represented for instance by the genome or proteome of complex organisms are immense, fast privacy preserving search algorithms are needed. In the previous work we introduced the foundation for such a privacy preserving genome search engine. In this work, we improve and elaborate on this and present an extensive evaluation and comparison showing that this scheme is both secure and practical. Our approach is based on Bloom filters with a configurable security property that performs more than 2000 times faster than PIR equivalents for large datasets, making it suitable for applications in bioinformatics. The results can then be further aggregated using Homomorphic Cryptography to allow an exact-match searching. In performance tests a search of a 50-nucleotides-long sequence against human chromosomes can be securely executed in less than 0.1 s on a 2.8 GHz Intel Core i7. We offer the entire system as an open source service for the community and offer ready-to-use REST as well as SOAP Web services.

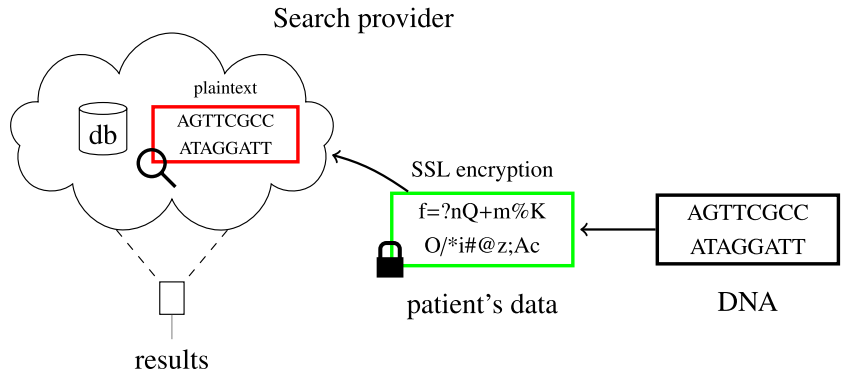
1. Introduction

Trust remains a crucial challenge concerning the outsourcing of computational tasks to the Cloud in the biomedical domain. Data can be transferred via encrypted channels, but as soon as processing begins, the data is disclosed to the Cloud provider that subsequently has reading access to the data and applies the processing algorithm to it in the clear (c.f. Fig. 1). This is a significant hindrance for many data intensive applications that could make good use of Cloud computing but have privacy and data protection requirements that forbid the disclosure of sensitive data to a third party. The search for DNA or a protein sequence in a database is an example of a biomedical application that uses patient related data that may allow re-identification of the subject. Gymrek et al. [1] describe how the re-identification of subjects is possible using their publicly available DNA data that had been thought of as anonymized.

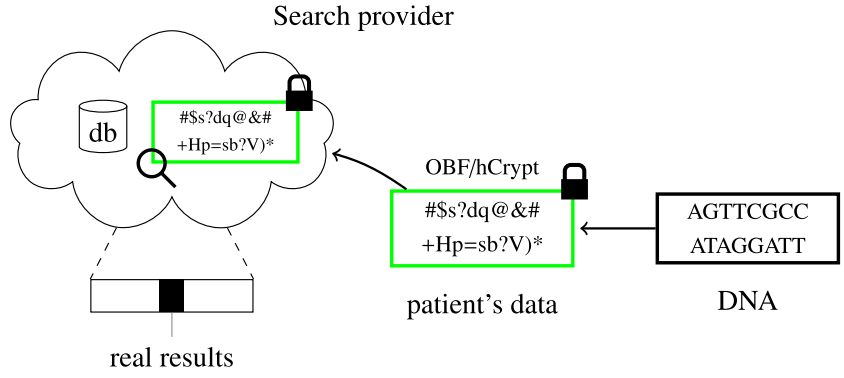
Homomorphic Cryptography (hCrypt, c.f. [2] for Gentry's breakthrough work) is a theoretically promising technology that provides a solution to this problem by allowing a remote party to execute arbitrary algorithms on encrypted data. This would allow the outsourcing of privacy constrained computational tasks to

* Corresponding author. Tel.: +49 511 762 79 58 69.
E-mail addresses: perl@dcsec.uni-hannover.de (H. Perl), y.mohammed@lumc.nl (Y. Mohammed), brenner@dcsec.uni-hannover.de (M. Brenner), smith@dcsec.uni-hannover.de (M. Smith).

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.12.006



(a) Encryption of transfer channels only.



(b) This article: obfuscation of the search query.

Fig. 1. Achieving privacy of patients' data.

Cloud resources without the need to establish trust, as the Cloud provider could not read the encrypted data. However, the performance of plain hCrypt for large data sets makes it unfeasible for most real world problems if used as a stand-alone solution, since recrypts are necessary after every AND gate and recrypts are a very expensive operation. Even in a recent scheme by Coron et al. [3], a recrypt operation takes 51 s for $\lambda = 62$ bits of security. Thus, the overhead from hCrypt housekeeping quickly dominates the real work.

The family of Private Information Retrieval (PIR) schemes establishes a cryptographic protocol that allows a user to search the database without revealing which item was queried. While some progress has been made in terms of runtime and communication complexity by Boneh et al. [4], Gentry and Ramzan [5], and Kushilevitz and Ostrovsky [6], searching the database is still at best bound linearly to the size of the database. For applications in biology and bioinformatics with large datasets, PIR schemes are still not feasible, as shown by our performance comparison.

1.1. Our contributions

Since hCrypt and PIR both solve the problem of private search in theory only, we have developed a Bloom filter based approach to privacy preserving search on databases [7] that allows for the feasible outsourcing of searching a sequence in a database by introducing a privacy/performance trade-off: instead of a hard security guarantee, the search result is hidden in noise. The ratio between real search and the noise can be configured so that it conforms to data protection regulations. The huge benefit of this somewhat weaker security guarantee is a performance increase of a factor of 2650 for the time it takes to execute a query compared to the recent PIR schemes (see Section 10.1). In detail, the runtime complexity of our scheme is in $\mathcal{O}(\log |A| + |s| + |R|)$ for a database A with the search term s and the results set R. This qualifies the algorithm to search large data sets.

The search algorithm operates on an Obfuscated Bloom filter (OBF) of the search term. In this paper, we prove that it is cryptographically hard to deduce the search term from the OBF or the results set. Furthermore, the additive property of Bloom filters is used to combine a set of queries into one that matches any search term of the set. This makes searching in a stream as efficient as searching in a set of discrete strings. We present a real world application of this principle from the biomedical domain and show that protected and secured search of encrypted DNA sequence queries in the complete human chromosomes is feasible. The output of this Bloom filter based search algorithm is then large enough to conform to the data protection regulations, as well as small enough to allow further processing using hCrypt.

In this article, we also present an analysis of how the Bloom filter search performs against Private Information Retrieval schemes on real-world datasets. In performance tests a search of a 50 nucleotides long sequence against human chromosomes can be securely executed in less than 0.1 s on a 2.8 GHz Intel Core i7. Finally, we show how the Bloom filter search can be integrated into the existing e-Science ecosystems as a Web service. We implement the Bloom filter search as a ready-to-use REST as well as a SOAP Web service.

1.2. Outline

In Section 2, we consider alternative methods to solve the problem of privacy preserving search possibilities and evaluate those against the Bloom filter search. In Section 3, a common notation for homomorphic encryption schemes and Bloom filters is established. Obfuscated Bloom filters are introduced in Section 4 as a crucial component to the search. In Section 5, we discuss the Bloom

filter search in detail and show a security analysis in Section 6. An implementation of the algorithms along with Web service interfaces is introduced in Section 8; a real world use case is analyzed in Section 9. We conclude with a performance evaluation and comparison with two PIR schemes in Section 10.

2. Related work

2.1. Private Information Retrieval (PIR)

In 2007, Boneh et al. [4] published a protocol that supports queries on encrypted data using a somewhat homomorphic crypto system by Boneh, Goh, and Nissim [8]. This crypto system allows for an arbitrary amount of additions, but only one multiplication. The system they present has a communication complexity of $O(\sqrt{A}\log^3 A)$ with A being the size of the database, whereas our approach only has a communication complexity of $\mathcal{O}(|s|)$ with s as the search term, as shown in Eq. (5). Therefore, the communication complexity of our scheme does not depend on the size of the database at all. Boneh et al. achieve a runtime complexity of $\mathcal{O}(|A| \cdot |s| \cdot \text{poly}\,\lambda)$ in combination with the PIR protocol by Cachin et al. [9]. In comparison, the total runtime complexity of our scheme is in $\mathcal{O}(\log|A| + |s| + |R|)$ for $|R|$ being the size of intermediate results produced by the Bloom filter search. Comparing security properties, Boneh's scheme completely hides the query from the server, whereas our scheme hides the query in $|R| = \binom{\lambda+k}{\lambda}$ queries (where k is the number of hash functions used in the Bloom filters and λ the obfuscation parameter). When setting $|R| \approx |A|$, we achieve the same security and runtime complexity as [4]. The parameter λ can be used to control the level of obfuscation and enables a security/performance trade off. This allows for the practical use of the system for a given amount of obfuscation (c.f. Sections 4 and 9).

Camenisch et al. [10] show how to integrate access permissions and policies into an oblivious transfer protocol by introducing a third party. This paper focuses on confidential access to public databases without access policies.

Canetti et al. [11] determine ways to provide verifiable results of delegated computation and information retrieval. Apart from the performance problem, this is another great challenge for future delegation scenarios but out of the scope of this paper.

2.2. Garbled circuits

The first secure approach to solve the problem of secret function evaluation was introduced in 1986 by Yao [12]. He presented the concept of *Garbled Circuits*, in which algorithms are translated to single-pass boolean circuits. The state tables of the gates in the circuit are then encrypted and shuffled within the state table, disguising the boolean function of a gate and essentially the function of the whole circuit.

Although this method is usable to some extent in an implementation by Malkhi et al. [13] and Malka [14], the core concept has some inherent deficits:

- the encryption and shuffling of the state tables introduce a dependency between the gates. This means that the circuit is only able to pass in one large pre-defined run. Modularity is not possible;
- furthermore, the security of the scheme relies on the fact that each gate only runs once. Therefore, only linear circuits are possible. Loops, for example, have to be unrolled completely;
- lastly, the input from the circuit creator is encoded right into the garbled circuit. In consequence, no memory access is possible, because the value of a memory cell cannot act as an input for the next cycle.

These restrictions lead to the fact that only limited algorithms can be transformed into a garbled circuit. Since the circuits can only be passed once, it is impossible to search twice with the same database and circuit.

2.3. Trusted Computing

In *Trusted Computing* [15], the integrity of the system is ensured through specialized hardware, the Trusted Platform Module (TPM). The TPM can measure the integrity of the bootloader, the operating system and software running in the operating system and help prevent tampered components from starting. The data and programs processed by the platform cannot be protected against the owner of the platform, as they are in possession of the root key for the platform. Therefore, Trusted Computing cannot be used to outsource searches if the database server is not trusted.

2.4. Encrypted CPU/hCrypt

Gentry [16,2] introduced a fully homomorphic crypto system. In this work he also stated that it is possible to evaluate circuits with arbitrary depth by mapping the mathematical operations to boolean operations, thus offering significant ground work for the encrypted execution of arbitrary programs. Based on this and the cryptosystem by Smart and Vercauteren [17] and Brenner et al. [18] developed an encrypted CPU capable of executing arbitrary programs by addressing issues such as memory access, branching and self-modifying code. With this CPU it is possible to execute arbitrary encrypted code as well as access an encrypted memory from within the code.

Although the simulated CPU solves the problem of secure function evaluation (SFE), it is still slower than the native code by several orders of magnitude. The following reasons can be identified:

- because of the oblivious memory access, the simulated CPU must re-evaluate all memory cells in every cycle, regardless of whether or not the plain text value actually changed;
- it is not possible to determine the end of a program without the knowledge of the secret key. Therefore when approximating the number of required cycles for the encrypted CPU, the worst-case runtime has to be assumed.

3. Preliminaries

In this section, we will introduce the basic components that are part of our search scheme as well as highlight their utility in the scheme.

3.1. Homomorphic encryption schemes

The discovery of a fully homomorphic encryption scheme by Gentry [16,2] in 2009 is an important foundation for the techniques described in this work. However, for the construction of our approach we do not require a particular instance of a homomorphic encryption scheme. Thus we use this generic definition:

Definition 1. A *fully homomorphic encryption scheme* $\mathcal{E}$ is a tuple of functions

$$(\mathsf{KeyGen}_{\mathcal{E}}() \mapsto (pk, sk), \mathsf{Encrypt}_{\mathcal{E}}(m, pk) \mapsto \mathfrak{c},$$
$$\mathsf{Decrypt}_{\mathcal{E}}(\mathfrak{c}, sk) \mapsto m, \mathsf{Evaluate}_{\mathcal{E}}(\mathfrak{c}_i, C, pk) \mapsto \mathfrak{c}')$$

with the following properties:

1. *Correctness of the encryption scheme:*

   $\mathsf{Decrypt}(\mathsf{Encrypt}(m, pk), sk) = m$

   for all outputs (pk, sk) of $\mathsf{KeyGen}()$.

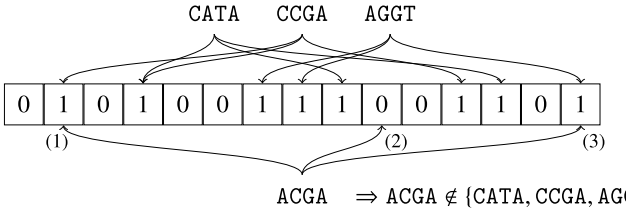


Fig. 2. Example of a Bloom filter for {CATA, CCGA, AGGT} with three hash functions.

2. *Correctness of the homomorphic property:*

$$\mathsf{Decrypt}\big(\mathsf{Evaluate}((\mathsf{Encrypt}(m_0, pk), \ldots, \mathsf{Encrypt}(m_n, pk)), C, pk), sk\big) = C(m_0, \ldots, m_n)$$

for all outputs (pk, sk) of $\mathsf{KeyGen}()$ and all boolean circuits $C : \{0,1\}^m \to \{0,1\}$.

For the remainder of this work we assume that all functions $(\mathsf{KeyGen}() \cdots)$ belong to the same encryption scheme, even without the index $\mathcal{E}$.

We set the plain text space P to be $\{0, 1\}$ and treat the cipher texts as encrypted bits for the boolean operations XOR and AND (which correspond to the addition and multiplication of two cipher texts). The cipher text space C of course depends on the particular scheme used.

In the search scheme, homomorphic encryption is used to execute an exact match search over the query and a reduced set of the database. Note that since the homomorphic encryption scheme is an *asymmetric* scheme, someone who wants to evaluate a circuit only needs the public key pk (along with the inputs) to compute $\mathsf{Evaluate}()$. Further, since $\mathsf{Encrypt}()$ can also be executed by the evaluator, the computations can also consist of a mix of plain text and ciphertext. The result would then be encrypted and retrievable only through the secret key sk.

3.2. Bloom filters

Bloom filters have been proposed by Bloom [19] in 1970. They provide a way to check whether a string is included in a set of strings, with a small probability that the Bloom filter wrongly outputs that the string is in the set while it is not (false positive). A Bloom filter is a bit array with a fixed length m. In order to create a Bloom filter from strings, one needs to also agree on a set of n hash functions that each has a target set corresponding to the indexes of the Bloom filter $\{0, \ldots, m\}$. For example, a Bloom filter for a string x is then created by hashing the string with each of the n hash functions and marking that index in the Bloom filter by writing a 1 at the position of the index. A Bloom filter for a set of strings $\{a, b, c\}$ is created by first creating the Bloom filters of the elements a, b, and c and then computing a component-wise boolean OR, i.e. a position in the Bloom filter is marked if that position was marked in any of the elements' Bloom filters. The creation of Bloom filters with size $m = 15$ and $k = 3$ hash functions for the set {CATA, CCGA, AGGT} is shown in Fig. 2.

In order to use the Bloom filters to check for a set membership (e.g. AGGT $\in$ {CATA, CCGA, AGGT}), the Bloom filters of an element AGGT and a set {CATA, CCGA, AGGT} are checked component-wise. If an index exists in the Bloom filters that is marked in the Bloom filter of the element AGGT but not in the Bloom filter of the set, then certainly AGGT is not included in the set (because the Bloom filter of the set is just a component-wise OR over the elements). If, however, every marked position in the Bloom filter of AGGT is also marked in the Bloom filter of the set, then AGGT is probably included in the set. This probability is bound by $(1 - e^{-k(|A|+0.5)/(m-1)})^k$ as shown by Goel and Gupta [20]. Going back to the example in Fig. 2, the string AGGT is not included in {CATA, CCGA, AGGT}, because hash function (2) for AGGT marked a position that was not marked for either CATA, CCGA, or AGGT.

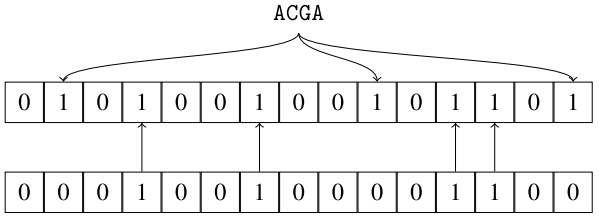


Fig. 3. A Bloom filter for ACGA with obfuscation parameter $\lambda = 4$.

Notation. We denote

- $\mathcal{B}^{k,m}(a)$ for the Bloom filter of a with size m and k hash functions,
- $\mathcal{B}^{k,m}(A)$ for the Bloom filter of a set A,
- $\mathcal{B}^{k,m}(a) \in \mathcal{B}^{k,m}(A)$ if $\mathcal{B}^{k,m}(a)_i \leq \mathcal{B}^{k,m}(A)_i$ for all i; and $\mathcal{B}^{k,m}(a) \notin \mathcal{B}^{k,m}(A)$ otherwise.

In this paper, Bloom filters are used to construct a binary tree to quickly reduce a large set of data to a small set of possible matches via a binary search. In order to search for a string x in a large database, we recursively check for the set membership of x in the set containing all words in the database and continue to check for the membership in the left and right half of the whole set if x is included in it.

4. Obfuscated Bloom filters

We extend Bloom filters to make false positives in the set membership check more probable. The motivation behind this is to pass an Obfuscated Bloom filter (OBF) of the query to the database that then finds all database entries that contain the OBF. Without obfuscation, this set would be small enough for the database to learn about the actual query. By using the OBF, we will show that the probability to retrieve the plain Bloom filter can be made small enough that the actual result is hidden in the large set of false positives, i.e. the noise; yet small enough to make the processing of these intermediate results in the homomorphic cipher text space feasible. Thus it can be ensured that only a configurable percentage of the results is not noise.

Obfuscating a Bloom filter works by incrementing λ random components of the Bloom filter $\mathbf{b} = \mathcal{B}^{k,m}(a)$, where λ is the obfuscation parameter. Fig. 3 shows a Bloom filter for a string ACGA that is obfuscated with $\lambda = 4$. The hamming weight of the OBF is always $k + \lambda$.

function obfuscate ($\mathbf{b}$, m, λ):
 for l *in* $[\lambda]$ {
  while ($i \xleftarrow{R} [m]$ and $b_i = 1$)
  $\mathbf{b}_i \leftarrow 1$
 }.

Because obfuscation marks λ additional positions in the Bloom filter, the set membership check needs some modification. Recall that the set membership is tested by checking for each position in the Bloom filter of the potential element whether that position is also marked in the set. Now that the potential element is obfuscated, the set membership is already concluded if at least k marked positions exist that are also marked in the set.

We now show that OBFs preserve the set membership property of plain Bloom filters.

Claim 1. *Given a Bloom filter* $\mathbf{B} = \mathcal{B}^{k,m}(A)$ *for a set* A *and a (plain) Bloom filter* $\mathbf{b} = \mathcal{B}^{k,m}(a)$ *as well as an obfuscated version* $\mathbf{b}' = \mathsf{obfuscate}(\mathbf{b}, m, \lambda)$. *Then:*

1. $\mathbf{b} \in \mathbf{B} \Rightarrow \mathbf{b}' \in_\lambda \mathbf{B}$
2. $\mathbf{b}' \in_\lambda \mathbf{B} \Rightarrow \Pr[\mathbf{b} \in \mathbf{B}] = 1/\binom{k+\lambda}{k}$.

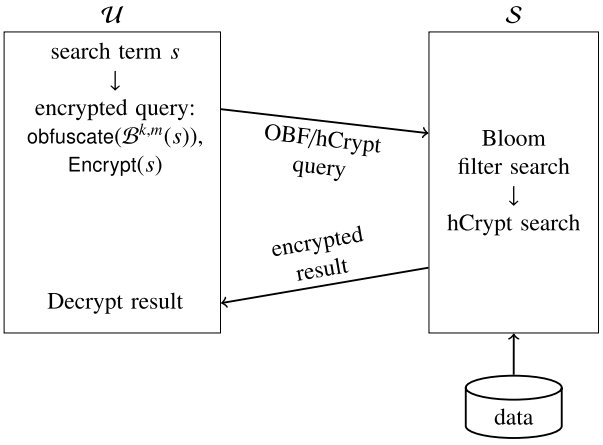


Fig. 4. Model for the search query.

Proof. Let $\mathbf{v} \leftarrow \mathbf{b}' - \mathbf{b}, \mathbf{v} \in \Delta^\lambda$ be the vector the Bloom filter $\mathbf{b}$ was obfuscated with.

1.
$$\begin{aligned}\mathbf{b} \in \mathbf{B} &\Rightarrow \forall i \in [m] : \mathbf{b}_i \leq \mathbf{B}_i \\ &\Rightarrow \forall i \in [m] : \mathbf{b}'_i - \mathbf{v}_i \leq \mathbf{B}_i \\ &\Rightarrow \forall i \in [m] : \mathbf{b}'_i \leq \mathbf{B}_i + \mathbf{v}_i \\ &\Rightarrow \forall i \in [m] : \exists \mathbf{v} \in \Delta^\lambda : \mathbf{b}'_i \leq \mathbf{B}_i + \mathbf{v}_i \\ &\Rightarrow \mathbf{b}' \in_\lambda \mathbf{B}.\end{aligned}$$
2. Given $\mathbf{b}' \in \mathbf{B}$, $\Pr[\mathbf{b} \in \mathbf{B}] = \Pr[\mathbf{b}' - \mathbf{v} \in \mathbf{B}]$ for the random vector $\mathbf{v}$ that $\mathbf{b}$ was obfuscated with. For a fix obfuscated vector $\mathbf{b}'$ (with $\sum_i \mathbf{b}'_i = k + \lambda$), there exist $\binom{k+\lambda}{\lambda}$ vectors $\mathbf{v}$ so that $\mathbf{v} = \mathbf{b}' - \mathbf{b}$. Therefore $\Pr[\mathbf{b} \in \mathbf{B}] = 1/\binom{k+\lambda}{\lambda}$. □

5. Bloom filter search

5.1. High-level view of the search scheme

Fig. 4 shows a general overview of how a search query is created and evaluated. User $\mathcal{U}$ wants to query Server $\mathcal{S}$ for a search term s without $\mathcal{S}$ learning about this search term or the result. First of all, $\mathcal{U}$ and $\mathcal{S}$ agree on a common alphabet Σ that influences the following parameters:

- the domain of each hash function used in the Bloom filters is Σ^*;
- the search term s of $\mathcal{U}$ is an element of Σ^* (a word of the alphabet Σ);
- the database A of $\mathcal{S}$ is a subset of $\mathcal{P}(\Sigma^*)$, the power set of Σ^* (a set of words of the alphabet Σ).

To execute a search, $\mathcal{U}$ transforms the *search term* into an *encrypted query*, which is an OBF of s.

This query is then sent to $\mathcal{S}$ and used in the *search algorithm*. The server uses the OBF to reduce the database to a smaller set of possible matches.

We formalize this protocol as follows.

1. *Setup of the Bloom filter tree.*
   $\mathcal{S}$ sets $T \leftarrow$ buildTree(root, database).
2. *Construction of the query.*
   The user $\mathcal{U}$ chooses a search term s and an hCrypt key pair $(pk, sk) \leftarrow$ KeyGen().
   He sets $\mathbf{b} \leftarrow$ obfuscate($\mathcal{B}^{k,m}(s), \lambda$) as well as $\mathfrak{s} \leftarrow$ Encrypt (s, pk).
3. *Transfer of the encrypted query.*
   $\mathcal{U}$ transfers $(\mathbf{b}, \lambda, \mathfrak{s}, pk)$ to $\mathcal{S}$.
4. *Bloom filter search.*
   $\mathcal{S}$ obtains a set of results $R \leftarrow$ searchTree($T, \mathbf{b}, \lambda$).
5. *hCrypt Search.*
   $\mathcal{S}$ obtains an encrypted result $\mathfrak{res}$ of the hCrypt algorithm on R and $\mathfrak{s}$.
6. *Transfer of the results.*
   $\mathcal{S}$ transfers $\mathfrak{res}$ back to $\mathcal{U}$, who then retrieves the final result $r \leftarrow$ Decrypt($\mathfrak{res}, sk$).

5.2. Setup of the Bloom filter tree

As mentioned above, a binary tree of Bloom filters is used to pre-filter the search results. Every node of the binary tree is a tuple $(\mathbf{b}, d, l, r)$, where $\mathbf{b}$ contains the Bloom filter associated with this node and l and r reference the left and right child of the node respectively. The l and r references are set to 'null' iff. the node is a leaf. In this case d holds the database entry associated with the leaf.

Now we can describe the algorithm that builds the binary tree from a database A. Note that this step is independent of the actual query and can therefore be calculated beforehand and used for many different queries.

```
1:  function buildTree(node, A):
2:      node.b ← B^{k,m}(A)
3:      if (|A| = 1) {
4:          node.l ← null
5:          node.r ← null
6:          return
7:      }
8:      buildTree(node.l, A|_{0..⌊|A|/2⌋})
9:      buildTree(node.r, A|_{⌊|A|/2⌋+1..|A|})
10:     return
```

The algorithm calculates the Bloom filter for the current node and then recursively calls itself on the left and right half of the database A when filling the left and right children of the node. Note that although the initial construction of the binary tree runs in time $O(|A|)$, updates to the binary tree are quite cheap, because only the nodes along the path from the changed node to the root need to be updated. Updates therefore run in time $O(\log |A|)$.

Fig. 5 shows the content of the binary tree after the construction.

5.3. Search using Bloom filters and binary search

Once the binary tree is constructed, the Bloom filter search is just a modified binary search over the Bloom filter of the query. In the following algorithm, *node* is initially the root node of the binary tree and $b =$ obfuscate($\mathcal{B}^{n,m}(s), \lambda$) is the OBF of the query.

```
1:  function searchTree(node, b, λ):
2:      if (b ∉_λ node.b) {
3:          return ∅
4:      }
5:      if (node.l = null or node.r = null) {
6:          return {node.d}
7:      } else {
8:          l ← searchTree(node.l, b, λ)
9:          r ← searchTree(node.r, b, λ)
10:         return l ∪ r
11:     }
```

Extension to stream search

For our application scenario we require the capability to search on streams, so we show how the Bloom filter search can be modified to find a substring in a stream (instead of a discrete set of words). Let s be the search term of $\mathcal{U}$ and A the database of $\mathcal{S}$, i.e. a character stream. The following steps are necessary to achieve a stream search:

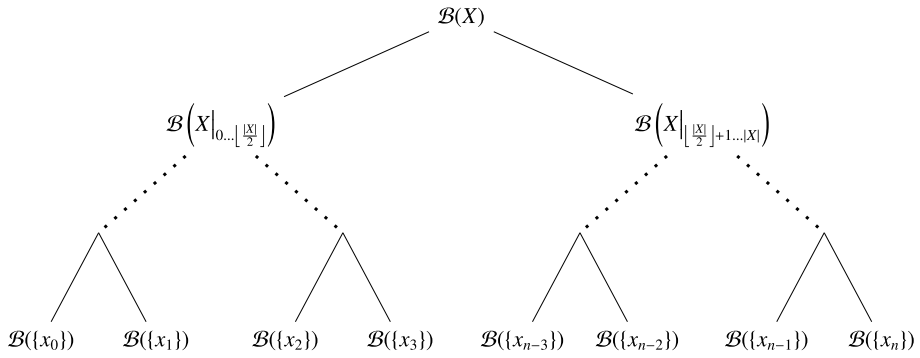
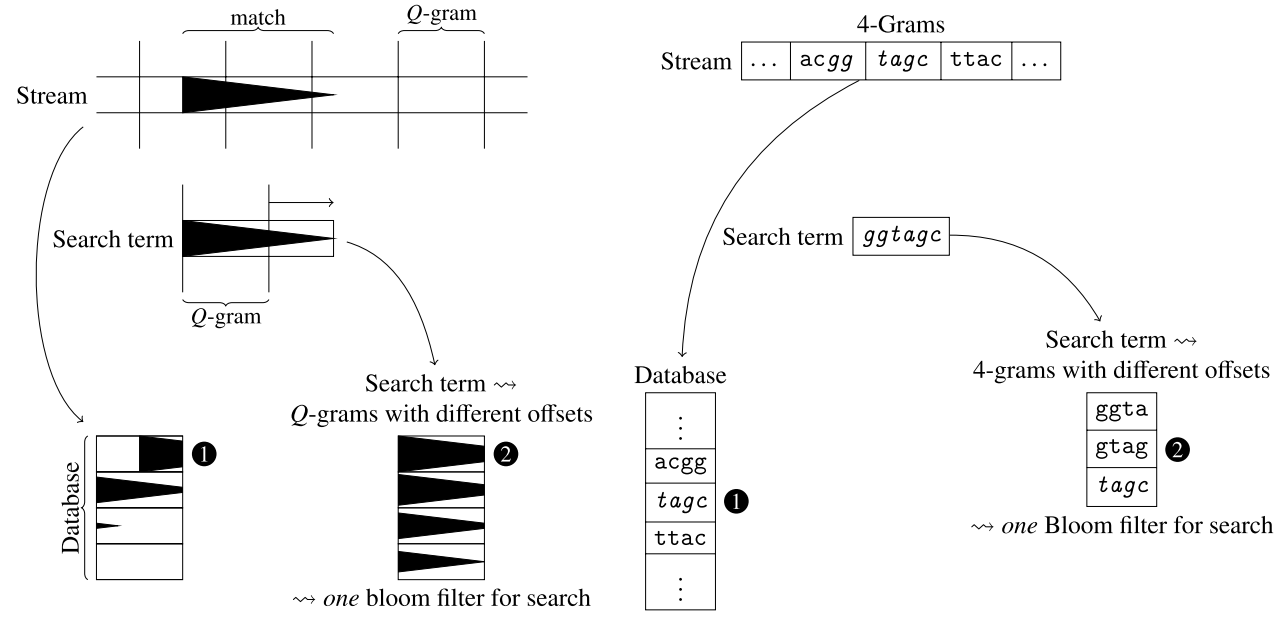


Fig. 5. Binary tree of Bloom filters.



(a) Schematic transformation.

(b) Transformation by example for the query "ggtagc".

Fig. 6. Transforming Bloom filter search to a stream search.

1. set the length of the Q-grams $Q \leq \lfloor |s|/2 \rfloor$. This guarantees that at least one Q-gram consists only of characters found in the search term;
2. split the stream into Q-grams, thus getting a database of roughly $|A|/Q$ words of size Q (1. in Fig. 6);
3. from the search term, generate $|s|$ Q-grams with offsets $0 \cdots |s| - 1$ (2. in Fig. 6);
4. combine the search terms Q-grams into one Bloom filter by adding them component-wise;
5. execute the search with one Bloom filter as described above;
6. for each result, do an exact-match comparison in the cipherspace, working directly on the encrypted search term and the stream;
7. compress the result in the cipherspace to return the encrypted positions of the search term in the stream.

This stream search extension is what we use to implement the real world use case in Section 9.

5.4. Exact search using hCrypt

After $\mathcal{S}$ executed the Bloom filter search described above, the (intermediate) results set R contains matches based on the OBF of the original search term s and may contain only a small percentage of real results. In order to send just these real results back to the user $\mathcal{U}$, hCrypt can optionally be used to execute an exact match search. This serves as an example of how the hCrypt can be used on the results of the Bloom filter search to add further processing to the whole search algorithm.

The goal of this step is for $\mathcal{S}$ to construct a single encrypted result $\mathfrak{res}$ that contains a small number of matches of the search query of $\mathcal{U}$. First, $\mathcal{S}$ constructs an indication vector $\mathfrak{ind}$ that marks all exact matches. Then, $\mathcal{S}$ uses this indication vector to filter just those intermediate results into the final results set which were marked before.

As all these transformations happen in the homomorphic cryptospace, $\mathcal{S}$ gains no knowledge over the final results during its construction.

5.4.1. Exact match search

First, $\mathcal{S}$ marks which results are actual matches of s in the homomorphic cipherspace. This is done by generating an encrypted bit-array $\mathfrak{ind}$ with the same size as the results set that will serve as an indication vector. More precisely let $\{r_i\}_{i=1}^{l} = R$ be the set of results. Then we want to construct an encrypted indication vector $\{\mathfrak{ind}_i\}_{i=1}^{l} \in C^l$ for the cipherspace C so that

$$r_i = s \Leftrightarrow \mathsf{Decrypt}(\mathfrak{ind}_i, sk) = 1 \quad \text{for all } i \in [l].$$

Note that in order to construct such a $\mathfrak{ind}$ in the homomorphic cipherspace, $\mathcal{S}$ only needs an encrypted version of s as well as the public key pk, which both can be sent to $\mathcal{S}$ as part of the encrypted query.

We assume that the plain text space $P = \{0, 1\}$. In order to compare words $\in \Sigma^*$ in the ciphertext space, words from the alphabet must first be transformed to a binary representation. Let $\mathsf{binary} : \Sigma^* \to \{0, 1\}^*$ be this function. This function can easily be constructed by first giving each character $c \in \Sigma$ a unique number and then encode a word $w \in \Sigma^*$ as a sequence of the padded binary representation of each character in w.

Let

$$\mathfrak{s} = \{\mathfrak{s}_j\}_{j\in[|s|]} \leftarrow \{\mathsf{Encrypt}(\mathsf{binary}(s)_j, pk)\}$$

be the encrypted search term and

$$\mathfrak{b} = \{\mathfrak{b}_{ij}\}_{i\in[l],j\in[|s|]} \leftarrow \{\mathsf{Encrypt}(\mathsf{binary}(r_i)_j, pk)\}$$

be the encrypted intermediate results set. The circuit for the character-level comparison is constructed as follows:

$$\mathfrak{ind}_i \leftarrow \bigwedge_j \mathfrak{b}_{ij} \oplus \mathfrak{s}_j \oplus 1 \quad \text{for all } i \in [l]. \tag{1}$$

Now that the indication vector is filled, the marked intermediate results need to be mapped to the final result.

5.4.2. Compressing the results

Assume that there are at most c entries in the vector $\mathfrak{ind}$ marked with 1. This number can be approximated from the number of results in the results set and the security parameter λ. Then an indication vector containing at most c marks can be expressed as a bit-vector with length $c \cdot |\mathfrak{s}|$. The basic idea is to shift each result which is a match (for which $\mathsf{Decrypt}(\mathfrak{ind}_i, sk) = 1$) by $|s|$ bits.

Let $\mathsf{hamming}(v)$ be the circuit that calculates the hamming weight of v as constructed by Smart and Vercauteren [17, p. 15]. This circuit returns a bit vector of length $\lceil \ln c + 1 \rceil$ if there are at most c 1's in v. Then, write down the cumulative sum of $\mathfrak{ind}$ as the $|R| \times \lceil \ln c + 1 \rceil$-matrix $(\mathfrak{sum}_{ij})$ using the following circuit:

$$\mathfrak{sum}_{ij} = \mathsf{hamming}(\mathfrak{ind}_{0\cdots i})_j \wedge \mathfrak{ind}_i. \tag{2}$$

Next, set the $|R| \times c$-matrix

$$\mathfrak{mask}_{ij} = \begin{cases} 1 & \sum_{k=0}^{\lceil c \rceil} 2^k \cdot \mathfrak{sum}_{ik} = j \\ 0 & \text{else.} \end{cases} \tag{3}$$

The final result can then be written as a $c \times |R|$-matrix $(\mathfrak{res}_{ij})$ with

$$\mathfrak{res}_{ij} = \bigoplus_{k=0}^{|R|-1} \mathsf{encrypt}(R_k)_j \wedge \mathfrak{mask}_{ki}. \tag{4}$$

Fig. 7 shows the steps for a small example. The rows are mirrored for better readability.

After this step, $\mathcal{S}$ holds an encrypted final results set $\mathfrak{res}$, which contains at most c matches of the database against the search term s. These results are then sent to $\mathcal{U}$, who decrypts $\mathfrak{res}$ using her private key sk.

6. Security analysis

For the search scheme to be secure one needs to show that an adversary is unable to extract the plain search query from the inputs, i.e. security against ciphertext-only attacks (COA) on the encrypted query. More precisely, given a tuple $q = (\mathbf{b} = \mathsf{obfuscate}(\mathcal{B}^{k,m}(s)), \lambda, \mathsf{Encrypt}(s, pk), pk)$ it is hard for an adversary $\mathcal{A}$ to retrieve s in probabilistic polynomial time (PPT). With that in mind we define the following security game:

Definition 2. For COA-security of the search scheme, consider the following game between a challenger $\mathcal{C}$ (which takes the role of the user in our scheme) and an adversary $\mathcal{A}$. The functions $\mathsf{KeyGen}()$, $\mathsf{Encrypt}()$, $\mathsf{Decrypt}()$ and $\mathsf{Evaluate}()$ are assumed to be indistinguishable under a chosen-plain text attack (IND-CPA secure). For example, the encryption schemes by Brakerski et al. [21] or Smart and Vercauteren [17] are all IND-CPA secure. The game consists of the following steps:

1. $\mathcal{C}$ chooses a random s, generates a key pair $(pk, sk) = \mathsf{KeyGen}()$, computes q as above and sends it to $\mathcal{A}$.
2. $\mathcal{A}$ can repeatedly ask for the encryption of a specific query l; $\mathcal{C}$ replies by sending $(\mathbf{b} = \mathsf{obfuscate}(\mathcal{B}^{k,m}(l)), \lambda, \mathsf{Encrypt}(l, pk), pk)$.
3. $\mathcal{A}$ computes a query s' and sends it to $\mathcal{C}$.
4. $\mathcal{C}$ outputs 1 iff. $s = s'$.

The adversary's advantage $\mathrm{Adv}_{\mathcal{A}}(\lambda)$ is defined as $\Pr[s = s']$. A search scheme is COA-secure if, for all $\mathcal{A} \in PPT$ the function $\mathrm{Adv}_{\mathcal{A}}(\lambda)$ is negligible.

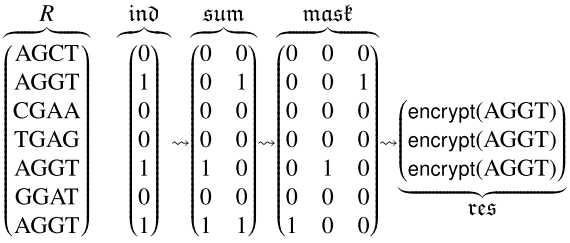


Fig. 7. Steps for compressing a results set. For the query "AGGT" the exact-match search circuit marked entries 2, 5 and 7 in $\mathfrak{ind}$, which is then transformed according to Eqs. (2)–(4).

Claim 2. *$\mathrm{Adv}_{\mathcal{A}}(\lambda)$ is a negligible function for any adversary that runs in probabilistic polynomial time.*

Proof. We show that $\mathrm{Adv}_{\mathcal{A}}(\lambda) \leq 1/\binom{k+\lambda}{\lambda}$, which is negligible in λ, by contradiction.

Assume that there exists an adversary $\mathcal{A}$ that runs in probabilistic polynomial time and breaks the security game with advantage $> 1/\binom{k+\lambda}{\lambda}$. Then we distinguish between three distinct cases:

1. $\mathcal{A}$ breaks the game by using $(\mathbf{b} = \mathsf{obfuscate}(\mathcal{B}^{k,m}(s)), \lambda)$ to retrieve s from $(\mathsf{Encrypt}(s, pk), pk)$.
   $\mathcal{A}$ can then be used to break the IND-CPA security of the underlying crypto system. Recall that the IND-CPA security game challenges an adversary $\mathcal{A}'$ who generated two plain texts (s_1, s_2) to decide whether a ciphertext c is an encryption of s_1 or s_2 with non-negligible advantage. This is how $\mathcal{A}'$ breaks the game using $\mathcal{A}$:
   (a) $\mathcal{A}'$ sends (s_1, s_2) to the challenger and receives a ciphertext c and a public key pk.
   (b) $\mathcal{A}'$ uses $\mathcal{A}$ to ask for the encrypted queries $q_i = \{(\mathbf{b})_i, \lambda, c_i, pk'\}_{i=1}^2$.
   (c) $\mathcal{A}'$ sets $q_i \leftarrow \{(\mathbf{b})_i, \lambda, c, pk\}_{i=1}^2$ and uses $\mathcal{A}$ to retrieve s_1', s_2' (which we assumed $\mathcal{A}$ can do).
   (d) $\mathcal{A}$ outputs i where $s_i' = s_i$.
   This contradicts the assumption that the underlying crypto system is IND-CPA secure.
2. $\mathcal{A}$ already breaks the game with inputs only $(\mathsf{Encrypt}(s, pk), pk)$.
   It is easy to see that this again gives an adversary $\mathcal{A}'$ that breaks CPA-security of the underlying homomorphic crypto system and therefore contradicts the assumption that the crypto system is IND-CPA secure. Note that this also covers the case where $\mathcal{A}$ uses $(\mathsf{Encrypt}(s, pk), pk)$ to retrieve s from $(\mathbf{b}, \lambda)$.
3. $\mathcal{A}$ already breaks the game with inputs only $(\mathbf{b}, \lambda)$.
   $\mathcal{A}$ can then construct a large set $A = \{a_i\}_{i=1}^l$ so that $\mathbf{b} \in_\lambda \mathcal{B}^{k,m}(a_i)$ for each i. Then starting from the assumption

$$\frac{1}{\binom{\lambda+k}{\lambda}} < \mathrm{Adv}_{\mathcal{A}}(\lambda)$$

we have the contradiction

$$\begin{aligned}\mathrm{Adv}_{\mathcal{A}}(\lambda) &= \Pr[s = a_i] \quad \text{(Definition of } \mathrm{Adv}_{\mathcal{A}}\text{)}\\ &\leq \Pr[\mathbf{b} \in_\lambda \mathcal{B}^{k,m}(a_i)] \quad \text{(Apply Bloom filters)}\\ &= \frac{1}{\binom{\lambda+k}{\lambda}} \quad \text{(Claim 1)}\\ \Rightarrow \frac{1}{\binom{\lambda+k}{\lambda}} &< \frac{1}{\binom{\lambda+k}{\lambda}}. \quad \square\end{aligned}$$

7. Choosing an obfuscation parameter

The real results to noise ratio is a measure of how unlikely captured data by an attacker would be useful. The smaller the ratio is, the harder an attacker can make any use of it. For instance hiding in 1% means the probability that any conclusions of an attacker about the original search term and the results will be wrong in 99% of the cases. So if a study associates one sequence S with a specific illness and we search for S on the genome with 1% real results to noise hiding ratio, it means that an attacker will have 99 wrong sequences along with S. An attacker can increase the certainty of her choices by including more reasoning knowing for instance which searches the user is interested in, what the user is attempting to publish, what results do not make sense for the user, and so on. Anyhow, taking into account that the real results to noise hiding ratio can be variable and is unknown to the attacker, it involves a huge effort from an attacker to reach a certainty that makes any conclusions statistically valid.

8. Implementation

8.1. Source code

The Bloom filter search algorithm was implemented in C as a binary tree with one root node and a pointer to the left and right child of the node. Each node contains information about the Bloom filter that represents the current subtree. Further, each leaf contains the corresponding database value.

The construction of the Bloom filter tree (i.e., indexing of the database) is done iteratively from a file. The algorithm scans the file to determine the number of nodes required and then builds the Bloom filter tree from the bottom up. The advantage over a naïve recursive implementation is that all memory allocation can be done in one call.

The source code can be found at https://hcrypt.com/downloads/bf-search.zip and builds using CMake. For compilation and linking, we require OpenSSL (for the hash functions in the Bloom filter tree) as well as Ruby 1.9 for the Web service (described below). Amongst others, the command-line program `bloom_search` is built, which takes as a mandatory parameter a chromosome file to index. In the path `/chromosomes` is a script `download.sh` that fetches all human chromosome files from http://hgdownload.cse.ucsc.edu/.

The implementation code for the Bloom filter search is organized in four parts: the main executable in `src/bloom_search`, the libraries for the Bloom filter and the Bloom filter tree in `src/lib/bloomfilter` and `src/lib/tree` respectively, and the ruby bindings in `src/binding`. Last but not least, unit tests can be found in the directory `src/test`. As the main executable does nothing else than parsing command line parameters and calling the libraries, we focus on those in the following sections.

8.1.1. Implementation of the Bloom filter

Internally, a Bloom filter is represented as an array of 64-bit integers. Each entry in the Bloom filter is then represented as a bit in one of the integers, so that a Bloom filter of size n can be represented as an array of size $\lceil \frac{n}{64} \rceil$. The functions bf_getpos() as well as bf_setpos() are used to access the individual entries without direct byte-arithmetics. Further, functions for creating a Bloom filter from a string (bf_hash()) and obfuscation (bf_obfuscate()) are included in this module.

8.1.2. Implementation of the Bloom filter tree

A Boom filter tree is an opaque struct that can be created by indexing a chromosome file (tree_index_file()). The struct contains a reference to the root node, the filename, and a list of matches which will be filled during the search. Indexing the database from a file has been parallelized using the pthreads library. The algorithms for indexing and searching have been discussed in detail in Sections 5.2 and 5.3.

8.2. Web service

The Bloom filter search library is written in C in consideration of speed and portability. In order to provide integration with various different infrastructures, we provide a wrapper around the library that exposes the search functionality through a Web service. The service understands *Representational State Transfer* (REST) as well as SOAP.

8.2.1. REST interface

Using REST, the Web service interface is exposed through HTTP verbs (GET, POST, PUT, PATCH, DELETE) that connect to a URI. The Web service is written in Ruby (as a rack-application) and can be found in the path `ws/rest-server/rest.rb`. The script expects a Ruby-C-extension built in `build/bftree.so` (which can be satisfied by using `build/` as the build directory) as well as the chromosomes downloaded in `chromosomes/`.

For the search service, only one endpoint is required: `GET/search`. The following parameters must be specified in the request:

`query`: The obfuscated Bloom filter, encoded as 0's and 1's.
`obfuscations`: The obfuscation parameter.

The service then replies with a JSON-encoded hash containing the query, the obfuscations, the time the search took, and an array of matches.

Example. The client sends the following message to the server:

```
http://hcrypt.com:1338/search?
obfuscations=10&
query=00000000000001001001010001000100001000
0100000101000000011011
00001000001000000000000000000000000100000000
0000000000100000100
```

The server responds with the following message:

```
{
  "query": "00000000000001001001010001000
  10000...",
  "time": 0.0500000007450581,
  "matches": [
    { "match": "AAGCT", "position": 7 },
    { "match": "ggctg", "position": 428050 },
    { "match": "GTTGC", "position": 54161548 }
  ]
}
```

8.2.2. SOAP interface

SOAP is still a widely-used standard for calling a Web service as well as orchestrating multiple Web services. Therefore, we developed a SOAP front end for the REST interface which can be called from any SOAP-aware program. The server resides in the path `ws/soap-server/GSearch/`. Additionally, an example SOAP

client can be found in `ws/soap-server/GSearchClient/`. Both client and server are written in Java. The WSDL description of the Web service can be found at http://www.hcrypt.com:1337/GSearch/services/GSearch?wsdl.

8.2.3. Clients

Amongst others, the obfuscation of the Bloom filter happens on the client side. Since the Bloom filters are used for the actual search, it is crucial that the client and the server construct the Bloom filters in the same way. In the following pseudo code, m is the size of the Bloom filter and k the number of hash functions. The hash functions are all derived from a SHA hash of the query.

```
1: function construct_bf(str):
2:     bf ← {0}_{i=0}^{m−1}
3:     d ← bytes(SHA1(str))
4:     for i in [k] {
5:         bf_{d_i mod m} ← 1
6:     }
7:     return true
```

For writing a new client, one could either use the `bf_hash()` function in the `libbloomfilter` library or use the client in `ws/client/`. The client is written in JavaScript and runs entirely in the browser. Communication with the REST Web service is done through asynchronous HTTP requests (AJAX). A live demo runs on https://hcrypt.com/bf-client/.

8.3. Asymptotic runtime and communication complexity

8.3.1. Communication complexity

Looking at the protocol introduced in Section 5.1, the only information sent from $\mathcal{U}$ to $\mathcal{S}$ is the tuple $q = (\mathbf{b}, \lambda)$. Splitting up the tuple into it is components, we get

- $|\mathbf{b}| = |\mathcal{B}^{k,m}|$ is in $\mathcal{O}(m)$, as $\mathbf{b}$ is a vector of length m,
- $|\lambda|$ is in $\mathcal{O}(\log \lambda)$, as λ has $\log \lambda$ bits in binary representation.

Because the encrypted query is just a concatenation of the components, $|q| \in \mathcal{O}(m + \log \lambda)$. The information sent from $\mathcal{S}$ to $\mathcal{U}$ is the results set R with the size $|R| = \binom{k+\lambda}{\lambda}$. Combining the total traffic, the overall combined complexity is in

$$\mathcal{O}\left(m + \log \lambda + \binom{k+\lambda}{\lambda}\right) \approx \mathcal{O}(|s|), \tag{5}$$

because the other parameters do not depend on the input of $\mathcal{U}$ but instead are parameters of the search scheme.

Note that the communication complexity depends only on parameters of the search protocol and not on the size of the database.

8.3.2. Runtime complexity

The Bloom filter tree search resembles a binary search with runtime complexity in $\mathcal{O}(\log |A|)$ for a database A, as shown in the pseudo code for searchTree(). For each step of the traversal of the Bloom filter tree, the set membership $\in_\lambda$ has to be computed. This can be done in $\mathcal{O}(m)$ (for Bloom filter size m) using the following algorithm:

```
1: function included_in(b, B, λ):
2:     for i in [m] {
3:         if (b_i > B_i) {                    // Check for mismatch
4:             λ ← λ − (b_i − B_i)             // use λ to make B larger
5:             if (λ < 0) {                    // ...until no tolerance is left
6:                 return false                // ...which concludes ∉
7:             }
8:         }
9:     }
10:    return true
```

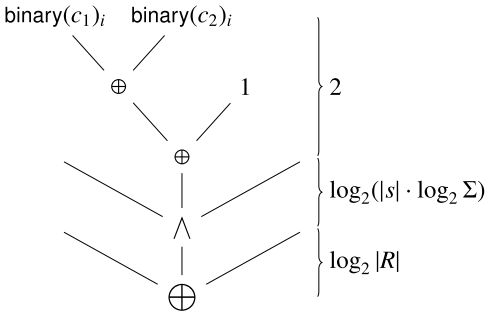


Fig. 8. Sketch of the search circuit.

This concludes that the runtime complexity of the Bloom filter search is in $\mathcal{O}(m \cdot \log |A|)$.

Next, we look at the complexity of the hybrid homomorphic part. The computational complexity of this part relates to the number of gates of the circuit, which can be split into two parts as follows.

1. *Construction of the indication vector* $\mathfrak{ind}$.
   This circuit has been constructed in Eq. (1). The inner term $(\mathfrak{b}_{ij} \oplus \mathfrak{s}_j \oplus 1)$ is conjugated over the binary representation of s and $|\text{binary}(s)| = |s| \cdot \log_2 |\Sigma|$. The conjunction can be expressed as a tree of binary AND-gates with $s_1 = 2 + (|s| \cdot \log_2 |\Sigma|)$ total gates.
2. *Construction of the final results set* $\mathfrak{res}$.
   In Eq. (4) the marked results are translated into the final results set. The XOR over all intermediate results R can again be written as a tree of binary XOR-gates, resulting in $s_2 = |R|$ total gates.

In Fig. 8 a schematic view of the depth of the circuit is shown. The final number of gates of the circuit is then given by

$$s_1 + s_2 = 2 + (|s| \cdot \log_2 |\Sigma|) + |R| \in \mathcal{O}(|s| + |R|), \tag{6}$$

again because the other parameters do not depend on the inputs of $\mathcal{U}$ or $\mathcal{S}$ but instead are parameters of the search scheme.

The size of R can be controlled by the parameters of the Bloom filter k and m as well as the security parameter λ.

Putting together the Bloom filter search and the homomorphic circuits, the total runtime complexity is in

$$\mathcal{O}(\log |A| + |s| + |R|) \tag{7}$$

when just considering the inputs from $\mathcal{U}$ and $\mathcal{S}$.

9. Use case

Moving from genetic to genomic research as well as the advances in the proteome research has resulted in sophisticated and large databases infrastructures, e.g. Ensembl (http://www.ensembl.org/), European Nucleotide Archive (ENA, http://www.ebi.ac.uk/ena/), or UniProt (http://www.uniprot.org/). Among different applications, these databases are used to perform the sequence alignment or sequences search. In the sense of moving towards personalized medicine as a strategic future healthcare paradigm, as well as pharmacogenetics and/or pharmacogenomics as a part of new drugs development process, performing an exact search of a specific nucleotide subsequence of a patient or subject in one or more of these genomic databases is essential. A key question regarding such search is the data protection and privacy constraints of the queried sequence. For instance, genetic/genomic sequences carry indication to different phenotypes. As not all genotypes–phenotypes correlations are known to us, outsourcing any database search that involves sending subject's genomics/genetic data outside the research institute becomes risky and violates the subject's privacy in many cases. This depends on the context of the subject consent, the country to where the data are outsourced and different other aspects. All of this makes outsourcing of the plain text sequence search difficult, and often forbidden due to privacy laws.

The hybrid homomorphic search introduced in this paper in cooperation with the bio-informatics lab of the Leiden University Medical Center allows us to operate outsourced searches in this environment without endangering the patient privacy.

9.1. Example

The following example should illustrate the application of the stream search to find a subsequence in a larger DNA sequence. For this purpose, let the large DNA sequence D with $|D| = 48$ be given by

$$D = \text{aggtcaagtccggaatacgtacgaacgtggcagctactcgagatccga} \tag{8}$$

and the search term s with $|s| = 8$ given by

$$s = \text{cgaacgtg}. \tag{9}$$

Next, choose $Q = 4 \leq \lfloor |s|/2 \rfloor$ and split D into twelve 4-grams

$D_0 = \text{aggt}$ $D_6 = \text{acgt}$
$D_1 = \text{caag}$ $D_7 = \text{ggca}$
$D_2 = \text{tccg}$ $D_8 = \text{gcta}$
$D_3 = \text{gaat}$ $D_9 = \text{ctcg}$
$D_4 = \text{acgt}$ $D_{10} = \text{agat}$
$D_5 = \text{acga}$ $D_{11} = \text{ccga}$

and build up the Bloom filter tree from these $\mathcal{B}^{k,m}(D_i)$'s. This completes the indexing phase on the server side.

From the query, generate five 4-grams

$s_0 = \text{cgaa}$ $s_3 = \text{acgt}$
$s_1 = \text{gaac}$ $s_4 = \text{cgtg}$
$s_2 = \text{aacg}$

and build one query Bloom filter $\mathcal{B}^{k,m}(\{s_i\})$. Note that this query has the same property as an obfuscated query (see Section 4) with an obfuscation parameter of $\lambda = 16$. The complete query q consists of $(\mathcal{B}^{k,m}(\{s_i\}), \lambda, \mathfrak{s}, pk)$ where $\mathfrak{s} = \text{Encrypt}(s)$ is the encrypted query. This completes the preparation phase on the client side.

The actual search is conducted using the Bloom filter search described above, using the set membership $\in_\lambda$. In this example, the results set R will include {(acgt, 4), (acgt, 6), (acgt, 9)}. This results set is transferred back to the client, who then checks each element of the results for an actual match.

10. Performance evaluation

As an example of real world problem sizes the procedure described above was used to index and search two human chromosomes of different sizes (available at ftp://hgdownload.cse.ucsc.edu/goldenPath/hg19/chromosomes/ or using the script in `chromosomes`/folder of the source code[1]). The experiments were run on a 2.8 GHz Intel Core i7 with 8 GB RAM. The index process utilizes all four hardware threads, the search runs only single-threaded. All timings are averages over 100 runs and can be reproduced by executing `script/benchmark.rb` in the source code package. The script expects that the Ruby library has been build in `build/release`.

Table 1 and Fig. 9 show the results for different human chromosomes. We chose Bloom filters of size 123, as 123 bits fit well into two native 64-bit integers with 5 bits left for internal flags, and 10 SHA-based hash functions to have enough room left for obfuscation as well as set-based Bloom filters. The percentage corresponds to the ratio of $\frac{\text{real results}}{\text{noise}}$. For example, hiding in 5% means that only 5% of the results are real, the other 95% of the results were noise added because of the Obfuscated Bloom filters. The baseline performance is given as a search with no obfuscation added.

Table 1
Performance of the obfuscated search.

	File size (MB)	Index (s)	Search (hiding in x%)			
			Baseline (s)	11% (s)	5% (s)	1% (s)
chr1.fa	254	3.62	0.016	0.056	0.078	0.096
chr2.fa	248	3.56	0.011	0.046	0.079	0.096
chr3.fa	201	2.74	0.009	0.038	0.063	0.076
chr4.fa	194	2.65	0.008	0.036	0.061	0.073
chr5.fa	184	2.50	0.008	0.034	0.056	0.070
chr6.fa	174	2.49	0.008	0.032	0.060	0.071
chr7.fa	162	2.24	0.007	0.030	0.049	0.061
chr8.fa	149	2.04	0.007	0.029	0.047	0.057
chr9.fa	144	2.10	0.005	0.023	0.039	0.047
chr10.fa	138	1.84	0.006	0.025	0.042	0.051
chr11.fa	137	1.82	0.006	0.025	0.042	0.051
chr13.fa	117	1.59	0.004	0.018	0.031	0.037
chr14.fa	109	1.48	0.004	0.016	0.028	0.034
chr15.fa	104	1.38	0.004	0.015	0.026	0.032
chr16.fa	92	1.22	0.004	0.015	0.025	0.031
chr17.fa	82	1.11	0.003	0.014	0.024	0.030
chr18.fa	79	1.04	0.003	0.014	0.024	0.030
chr19.fa	60	0.79	0.002	0.010	0.018	0.022
chr20.fa	64	0.85	0.003	0.011	0.019	0.023
chr21.fa	49	0.67	0.001	0.007	0.012	0.015
chr22.fa	52	0.72	0.001	0.006	0.011	0.014

Table 2
Performance of homomorphic post-processing.

Searchterm size (Bit)	Size of the set	
	100 elements (s)	1000 elements (s)
5	0.007	0.067
16	11.300	115.000
32	27.800	288.000
64	55.600	560.000

It should be noted that using our approach the database containing the Chromosomes only needs to be indexed once independently of the number of users querying the database.

As stated above, these results can be processed further using hCrypt due to the small size of the results generated by the Bloom filter search. The performance figures of an exact match search using hCrypt for different results set and search term sizes are outlined in Table 2.

10.1. Performance comparison with PIR schemes

For a performance comparison between our Obfuscated Bloom filter search and Private Information Retrieval (PIR) techniques, data from Costea et al. [22] will be used. Costea et al. have implemented two PIR schemes: one based on the *Quadratic Residuosity Assumption* (QRA-PIR) by Kushilevitz and Ostrovsky [6], and one based on the ϕ*-hiding Assumption* (ϕ-PIR) by Gentry and Ramzan [5]. Both implementations were written in C++, which is roughly comparable to the C implementation of the Bloom filter search.

The PIR implementations operate on the location-based data, but the general search scenario is the same: a private search for a point of interest (POI) in a database. We assume that a POI has a size of 32 B, i.e. it just contains the GPS coordinates. Fig. 10 shows the performance comparison of the two PIR schemes along with the performance of the Obfuscated Bloom filter search. Because the differences between the compared systems are so huge, the numbers can also be seen in Table 3. Since Costea et al. did not publish their source code for the PIR schemes, we relied on their measurements,

[1] https://hcrypt.com/downloads/bf-search.zip.

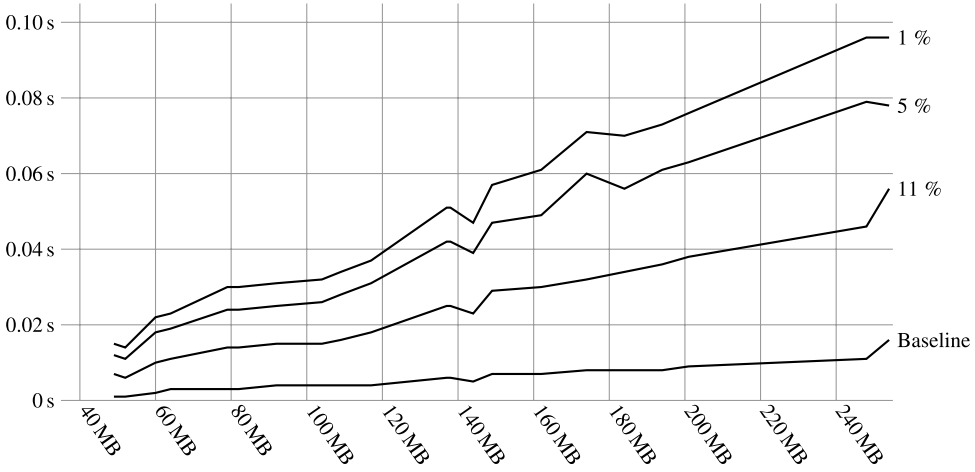
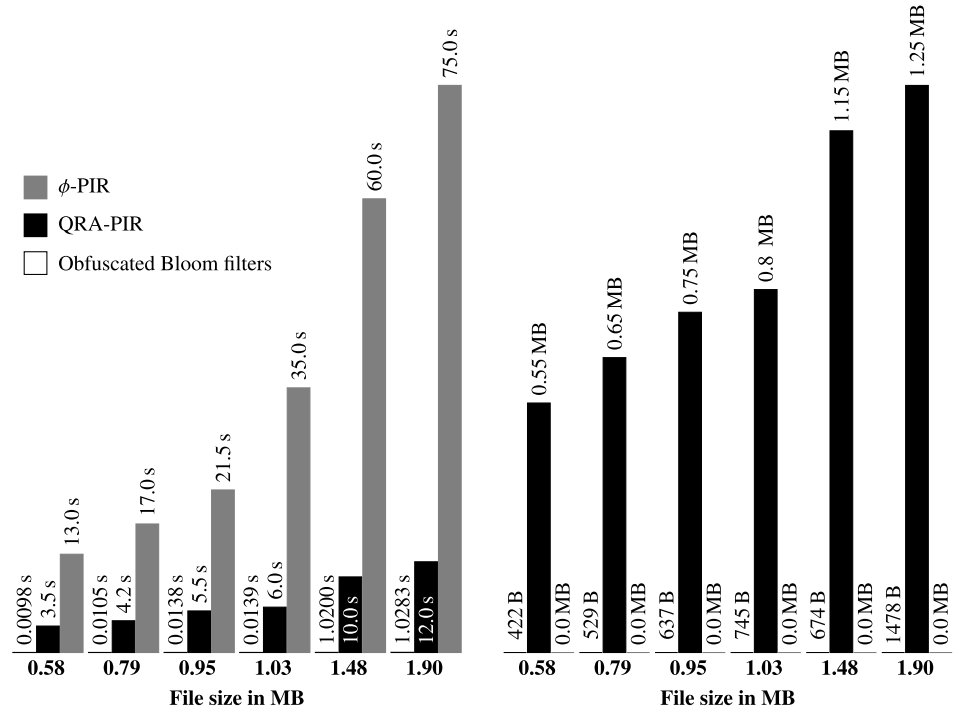


Fig. 9. Plot of the performance of the obfuscated search.



(a) Comparison of server CPU time.

(b) Comparison of communication size.

Fig. 10. Comparison of the Bloom filter search with two different PIR schemes.

which we took from the graphs in [22, Figs. 3–4]. For the communication size of ϕ-PIR, the graphs did not show a significant deviation from zero, which is why we set the column to 0.0 MB. Still, at least a single result would need to be transmitted.

For the server CPU timings, OBF is significantly faster than both PIR schemes, even when hiding in 1% of the results. Search times of 12 s for QRA-PIR and even 75 s for ϕ-PIR for a file under 2 MB size shows that even recent PIR schemes are still far from achieving the performance needed to search in large datasets common in biology. Considering the communication size, OBF is about as good as ϕ-PIR, and a drastic improvement over QRA-PIR (OBF: 1478 B, QRA-PIR: 1.25 MB). When combining the server CPU time and the communication size, the compared PIR schemes are either fast on the server or light on the wire, but not both. However, PIR still promises a stronger security, yet without the possible trade-off to sacrifice a little bit of security for a large performance boost.

Table 3
Comparison of the Bloom filter search with two different PIR schemes.

File size	Server CPU (time)			Communication (size)		
	QRA (s)	ϕ-PIR (s)	OBF (s)	QRA (MB)	ϕ-PIR (MB)	OBF (B)
0.58	3.5	13.0	0.0098	0.55	0.0	422
0.79	4.2	17.0	0.0105	0.65	0.0	529
0.95	5.5	21.5	0.0138	0.75	0.0	637
1.03	6.0	35.0	0.0139	0.80	0.0	745
1.48	10.0	60.0	0.0200	1.15	0.0	674
1.90	12.0	75.0	0.0283	1.25	0.0	1478

11. Conclusion

In this work, we introduced a search algorithm that utilizes Obfuscated Bloom filters to ensure the confidentiality of search queries as well as the results of the search. Through obfuscation, the query

can be made secure enough to conform to data protection regulations, yet the set of results is small enough to allow further processing with hCrypt, e.g. an exact-match search as demonstrated in this paper. Our approach achieves a communication complexity of $\mathcal{O}(|s|)$ as well as the runtime complexity of $\mathcal{O}(\log |A| + |s| + |R|)$ with a flexible parameter to adjust the size of the results set R.

Also, the size of the database that can be searched is sufficiently large and the search itself is fast enough ($\mathcal{O}(\log |A|)$) for first real world use cases to be practically implemented. We showed an extensive performance analysis and comparisons with PIR schemes. We presented a security analysis of our design and demonstrated its feasibility using datasets containing human chromosomes. This is one of the first systems to enable the practical use of hCrypt. Beyond offering a solution for search with encrypted terms, this paper also serves as an example of how systems can be designed to incorporate the new possibilities of Homomorphic Encryption.

References

[1] M. Gymrek, A.L. McGuire, D. Golan, E. Halperin, Y. Erlich, Identifying personal genomes by surname inference, Science 339 (6117) (2013) 321–324.
[2] C. Gentry, A fully homomorphic encryption scheme, Ph.D. Thesis, Stanford University, 2009.
[3] J.-S. Coron, D. Naccache, M. Tibouchi, Public key compression and modulus switching for fully homomorphic encryption over the integers, in: EUROCRYPT'12: Proceedings of the 31st Annual International Conference on Theory and Applications of Cryptographic Techniques, Springer-Verlag, 2012.
[4] D. Boneh, E. Kushilevitz, R. Ostrovsky, W.E. Skeith, III, Public key encryption that allows PIR queries, in: Proceedings of the 27th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO'07, Springer-Verlag, 2007.
[5] C. Gentry, Z. Ramzan, Single-database private information retrieval with constant communication rate, in: Automata, Springer, Berlin, Heidelberg, 2005, pp. 803–815.
[6] E. Kushilevitz, R. Ostrovsky, Replication is not needed: single database, computationally-private information retrieval, in: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, 1997. 1997, pp. 364–373.
[7] H. Perl, Y. Mohammed, M. Brenner, M. Smith, Fast confidential search for bio-medical data using bloom filters and homomorphic cryptography, in: Proceedings of the 8th IEEE International Conference on eScience, IEEE, 2012.
[8] D. Boneh, E.-J. Goh, K. Nissim, Evaluating 2-DNF Formulas on Ciphertexts, in: Lecture Notes in Computer Science, vol. 3378, Springer, Berlin, Heidelberg, 2005. Theory of cryptography edn.
[9] C. Cachin, S. Micali, M. Stadler, Computationally private information retrieval with polylogarithmic communication, in: Advances in Cryptology—EUROCRYPT'99, Springer, Berlin, Heidelberg, 1999, pp. 402–414.
[10] J. Camenisch, M. Dubovitskaya, G. Neven, Oblivious transfer with access control, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, ACM Request Permissions, 2009.
[11] R. Canetti, B. Riva, G.N. Rothblum, Practical delegation of computation using multiple servers, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, ACM Request Permissions, 2011.
[12] A.C.-C. Yao, How to generate and exchange secrets, in: 27th Annual Symposium on Foundations of Computer Science, 1986, 1986, pp. 162–167.
[13] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, Fairplay—a secure two-party computation system, USENIX Security Symposium.
[14] L. Malka, VMCrypt: modular software architecture for scalable secure computation, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, ACM Request Permissions, 2011.
[15] C. Mitchell, Chris (Eds.), Trusted Computing, Institution of Engineering and Technology, 2005.
[16] C. Gentry, Fully homomorphic encryption using ideal lattices, in: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC'09, ACM Request Permissions, 2009.
[17] N. Smart, F. Vercauteren, Fully homomorphic encryption with relatively small key and ciphertext sizes, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pp. 420–443.
[18] M. Brenner, J. Wiebelitz, G. von Voigt, M. Smith, Secret program execution in the cloud applying homomorphic encryption, in: 2011 Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference, DEST, 2011.
[19] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.
[20] A. Goel, P. Gupta, Small subset queries and bloom filters using ternary associative memories, with applications, in: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '10, ACM Request Permissions, 2010.
[21] Z. Brakerski, C. Gentry, V. Vaikuntanathan, (Leveled) fully homomorphic encryption without bootstrapping, in: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ITCS'12, ACM Request Permissions, 2012.
[22] S. Costea, D.M. Barbu, G. Ghinita, R. Rughinis, A comparative evaluation of private information retrieval techniques in location-based services, in: 4th International Conference on Intelligent Networking and Collaborative Systems, INCoS, 2012.

H. Perl received his Master's degree in computer science in December 2011 from the Leibniz University Hanover, Germany and joined the university's Distributed Computing & Security Group in January 2012 as a doctorate student. While he was still a graduate student he developed the first open-source homomorphic cryptography library. His research interests include homomorphic cryptography and its application in Cloud Computing.

Y. Mohammed obtained his Ph.D. in Medical Informatics from the University of Göttingen. Since 2008 he is a postdoctorate member of the Distributed Computing and Security group at the University of Hannover and since 2011 he is affiliated with the High Throughput Proteomics group at Leiden University Medical Center (LUMC). The fields of Yassene's work and research have been biomedical signal and image processing, modeling and simulation of minimal invasive surgery, as well as healthgrids and cloud computing. Currently his main focus is computational high throughput proteomics and using scientific workflow for processing biomedical big data, all with the accompanying data protection issues.

M. Brenner is currently working as a research associate at Leibniz University Hannover, Germany. He received his Master's degree in computer science from the University of Applied Sciences in Hannover in 2007. Between 1995 and 2008 he was also employed in the financial industry as a developer and software architect for distributed systems in IBM mainframe/Sun/JEE-based infrastructures. In 2012 he received his Ph.D. His current work focuses on the security in distributed systems, applications and architectures for encrypted program execution and encrypted circuit design. He is a member of the IEEE.

M. Smith is a Professor of Computer Science at the Leibniz Universität Hannover, Germany where he leads the Distributed Computing & Security Group. He completed his studies of Computer Science & Electrical Engineering at the University of Siegen, Germany with distinction. Subsequently he was a full time researcher at the Philipps Universität Marburg, Germany where he completed his Ph.D. in 2008, also with distinction. In 2009 he was awarded the Ph.D. Prize for outstanding innovation by the Gesellschaft zur Förderung des Forschungstransfers (GFFT e.V.). His current research is focused on the usability aspects of security and privacy mechanisms with a wide range of application areas. These areas include Cloud computing, e-Research infrastructures, Social Networking and Mobile Computing. He is a member of the IEEE and the ACM SIGSAC.