Pre-Proceedings

Workshop on Encrypted Computing & Applied Homomorphic Cryptography

WAHC’16

In association with Financial Cryptography & Data Security 2016

Rockley Christ Chuch, Barbados
**Program Committee**

- Dan Bogdanov, Cybernetica, Estonia
- Marten van Dijk, UConn, USA
- Joan Feigenbaum, Yale, USA
- Rosario Gennaro, CCNY, USA
- Sergey Gorbunov, MIT, USA
- Aggelos Kiayias, UConn, USA
- Vlad Kolesnikov, Bell Labs, USA
- Kim Laine, Microsoft, USA
- Tancrède Lepoint, CryptoExperts, France
- David Naccache, ENS, Paris, France
- Michael Naehrig, Microsoft, USA
- Pascal Paillier, CryptoExperts, France
- Benny Pinkas, Bar-Ilan University, Israel
- Yuriy Polyakov, NJIT, USA
- Berk Sunar, WPI, USA
- Mehdi Tibouchi, NTT, Japan
- Yevgeniy Vahlis, Amazon, USA
- Fré Vercauteren, KU Leuven, Belgium
- Adrian Waller, Thales, UK
## Content

4  Gamze Tillem, Ömer M. Candan, Erkay Savas and Kamer Kaya:  
*Hiding Access Patterns in Range Queries using Private Information Retrieval and ORAM*

20 Liisi Kerik, Peeter Laud and Jaak Randmets:  
*Optimizing MPC for robust and scalable integer and floating-point arithmetic*

36 Yarkın Doröz, Gizem Cetin and Berk Sunar:  
*On-the-fly Homomorphic Batching/Unbatching*

50 Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin Butler and Patrick Traynor:  
*Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation*

66 *CallForFire: A Mission-Critical Cloud-based Application Built Using the Nomad Framework*
Hiding Access Patterns in Range Queries using Private Information Retrieval and ORAM

Gamze Tillem¹, Ömer Mert Candan¹, Erkay Savas¹, and Kamer Kaya¹
Sabanci University, Istanbul, Turkey
{gtillem, mcandan, erkays, kaya}@sabanciuniv.edu

Abstract. We study the problem of privacy preserving range search that provides data, query, and response confidentiality to the users for range queries. We propose two methods based on Private Information Retrieval (PIR) and Oblivious RAM (ORAM) techniques. For PIR-based queries, Lipmaa’s computationally-private information retrieval (CPIR) scheme is employed. For the ORAM-based method, Stefanov et al.’s Path ORAM scheme is adapted to enable privacy preserving range search. Our analyses show that from the computational point of view, the ORAM-based method performs much better due to cheap server operations. However, CPIR utilizes the bandwidth better especially for large databases, its security definitions are more formal, and it is more flexible for various settings with multiple clients and/or bandwidth limitations. In this work, to make CPIR a practical alternative for large databases, we improve its performance via shared memory OpenMP and distributed memory OpenMP-MPI parallelization with a scalable data/task partitioning.

Keywords: Privacy preserving range queries, private information retrieval, oblivious RAM, data privacy, parallel computing.

1 Introduction
While outsourcing the data storage to cloud is beneficial for data owners to reduce the associated costs, ensuring a secure and private access to it becomes the next big challenge. The threat is that a curious data-holder may try to retrieve information from the stored data or the results of the queries sent by the data owner. Therefore, several approaches have been proposed in the literature to securely search over outsourced data for a specific item or for multiple items in a range. Existing approaches for range queries on encrypted data include encryption techniques that preserve the order of plaintext values [1], use of predicate functions based on cryptographic properties [2], utilizing special data structures [10], and using a bucketization method [5] based on data partitioning.

Regardless of the approach used, a privacy preserving range query scheme needs, in general, to deal with three security issues: providing data confidentiality, providing query confidentiality, and preventing the disclosure of the query access patterns. Data confidentiality is guaranteed by encryption in existing privacy preserving range query methods. The confidentiality of a query is provided by its transformation into a secure representative. But, almost none of the existing methods aims to hide the query access patterns, since the solution of the problem requires the use of computationally expensive schemes such as Private
Information Retrieval or Oblivious RAM. Nonetheless, recent advances in the literature such as the fast Path ORAM method of Stefanov et al. [7], yield almost practical schemes to enable hiding query access patterns. Similarly, certain acceleration techniques for PIR yield significant performance improvements [6,9].

The main contribution of this work is to explore the feasibility of hiding access patterns in secure and private range queries. We introduce two techniques: one based on CPIR [6], and the other on Path ORAM [7] and compare them in terms of communication and computation costs. For the CPIR-based scheme, we devise novel parallelization approaches in both shared and distributed memory settings. Although the CPIR-based scheme is computationally less efficient than the Path ORAM-based one, as our analyses show, a practical, parallel CPIR-based implementation is an important contribution since Path ORAM has a significant bandwidth usage and it is not as flexible as CPIR for various settings such as one with multiple clients.

The outline of the paper is as follows: Section 2 gives the background. In Section 3, multi-dimensional privacy-preserving range query methods using PIR and Path ORAM are explained in detail. The security of proposed methods are explained in Section 4. Section 5 provides a complexity analysis of the two approaches in terms of their communication and computational complexities. Experimental results to compare these two approaches are given in Section 6. Section 7 concludes the paper.

2 Background

Our work is based on three main concepts: Lipmaa’s BddCPIR model [6], Stefanov’s Path ORAM model [7], and Hore’s bucketization model [5]. Here we explain these concepts.

2.1 Privacy preserving range queries using bucketization

In bucketization, a secure index tag for each data item is generated using a predefined rule and the data items are partitioned to a bucket depending on this tag [4]. The data buckets are stored in encrypted form in the database and a query is first translated into the corresponding bucket ids (by the client). Then, the data items with the matching bucket ids are retrieved from the database. If the server knows the matching bucket ids and has some information on some of the items, the privacy of the query can be disrupted. To obfuscate the retrieved data range and increase the security level, false positives are used within each bucket. However, the existence of false positives creates a performance overhead.

Hore et al. [5] proposed an algorithm which aims to generate optimized buckets in terms of performance and security; it starts with a greedy multi-partitioning phase which initially puts all the items in a single bucket and iteratively partitions an existing bucket into two until a desired number of buckets are obtained. At each iteration, a compactness function is used to decide the bucket to be partitioned. This approach reduces the number of false positives within a bucket; then a second algorithm, known as controlled diffusion [5], is applied on the buckets to redistribute bucket contents based on a pre-defined
degradation factor and thus, calibrates the number of false positives within each bucket for an acceptable security level.

The methods introduced in this paper do not solely depend on a controlled diffusion mechanism since the server cannot uniquely identify the retrieved buckets. Especially for the CPIR-based approach, the retrieved bucket can be an arbitrary one in the database from the server’s point of view.

2.2 Lipmaa’s BddCPIR protocol for PIR

The BddCPIR scheme [6] employs the Damgård-Jurik cryptosystem [3] which is based on the hardness of the decisional composite residuosity problem. To improve the efficiency in terms of communication and computation, the scheme employs binary decision diagrams.

**Damgård-Jurik Cryptosystem:** While the setting of the homomorphic Damgård-Jurik (DJ) cryptosystem is similar to that of RSA which also employs two sufficiently large primes $p$ and $q$ to work with a composite modulus $N = pq$; the security of the scheme is based on the hardness of decisional composite residuosity rather than integer factorization. The key property of DJ cryptosystem is a positive integer $s$ which provides block-length adjustment capability for the encryptions in different levels of Lipmaa’s BddCPIR. The DJ scheme is summarized in Figure 1.

### Key generation:
- Choose large primes $p$ and $q$, and set $N = pq$.
- Choose $g \in \mathbb{Z}_{N+1}^*$ s.t. $g = (1 + N)^j x \mod N^{s+1}$ with $j$ is a known relative prime of $N$ and $x \in H$, where $H$ is isomorphic to $\mathbb{Z}_N^*$.
- Compute $\lambda = \text{lcm}(p - 1, q - 1)$, and choose $d$ s.t. $d \equiv 1 \mod N$ and $d \equiv 0 \mod \lambda$.
- Public parameters: $(N, g)$
- Private parameters: $d$

### Encryption:
- $E(m, r) = g^m r^N \mod N^{s+1}$, where plaintext $m \in \mathbb{Z}_N$ and $r \in \mathbb{Z}_{N+1}^*$ is a random integer.

### Decryption:
- Compute $e^d \mod N^{s+1}$ and using the algorithm proposed in [3] find $m$.

### Additive homomorphic properties:
- $E(m_1)E(m_2) = E(m_1 + m_2)$
- $E(m)^r = E(mc)$

**Fig. 1.** Damgård-Jurik Cryptosystem

**Binary Decision Diagram (Bdd):** Bdds can be considered as binary trees. The internal nodes of a binary tree are represented as $R_{i,j}$, where $i$ is the level of the node in the tree and $j$ is the location of the node in the current level. The leaf nodes of the binary tree hold the data items. Each leaf node is represented as $f_x$ such that $x$ is an $m$-bit string that represents the route taken from the root to $f_x$ for a tree with height $m+1$. Figure 2 demonstrates an example of Bdd with 4 child nodes where 0 and 1 are used for the left and right child, respectively.

**2,1-CPIR protocol:** The client inputs either 0 or 1 to retrieve one of the 2 files ($f_0$ or $f_1$) stored in the server without leaking information. The flow of the protocol is as follows:
Client generates public and private keys \((pk, sk)\). She then chooses an \(x \in \{0, 1\}\), computes the encrypted selection bit \(c = E_{pk}(x)\), and sends \(pk\) and \(c\) to the server.

- Server computes \(R = E_{pk}(f_0)c_{f_1}c_{f_0}\) and sends \(R\) to the client.
- Client decrypts \(R\) with his private key \(sk\) to find \(f_x\).

From \((2, 1)\)-CPIR to \((n, 1)\)-CPIR: The 1-out-of-2 CPIR protocol can be extended to 1-out-of-\(n\) using a Bdd. the extended protocol starts with the leaf nodes of the tree and iteratively merges siblings until the root is reached. For each merge, an \(R\) value is calculated. The final result for the tree root is then sent to the client. In \((n, 1)\)-CPIR, the client needs to send \(m\) selection bits as \(x = (x_0, x_1, \ldots, x_{m-1})\) to denote the path. Furthermore, the result should be decrypted \(m\) times to retrieve the requested file. Figure 3 illustrates \((n, 1)\)-CPIR protocol for a 4-file case based on the binary tree in Figure 2.

**Client:**
- Encrypt the selection bits \(c_0 = E(x_0)\) and \(c_1 = E(x_1)\) and send them to the server.

**Server:**
- For the lowest level of the tree compute:
  - \(R_{1,0} = E(f_0)c_{f_1}c_{f_0}\)
  - \(R_{1,1} = E(f_2)c_{f_3}c_{f_2}\)
- For the next level repeat the computations; this time use \(R_{1,j}\) instead of the files and the encrypted selection bits of the current level:
  - \(R_{2,0} = E(R_{1,0})c_{R_{2,1}c_{R_{2,0}}}\)
- Send \(R_{2,0}\) to the client.

**Client:**
- Apply double decryption to \(R_{2,0}\) to retrieve the selected file.

Fig. 3. An example of \((n, 1)\)-CPIR for a database of 4 files

### 2.3 Path ORAM

Path ORAM [7] is a simple Oblivious RAM protocol used to prevent the leakage of access patterns to the outsourced data. In each access a full path of data is retrieved by client and it is written back to database after shuffling and re-encryption. The details of the protocol are as follows:

- The server stores the data in a binary tree structure. Each node of the tree is called a bucket. In each bucket, \(Z\) blocks of data are stored. If a bucket has less than \(Z\) blocks, dummy blocks are added. At the beginning, all buckets are initialized with some dummy values.
The client maintains a local stash, a small and private storage, to perform shuffling and re-encryption operations on the accessed data path. She also maintains a position map that gives the current location of a data item. At the beginning of the protocol, the stash is empty and the position map assigns data items into some random buckets.

Access protocol for read and write operations: To read/write data, the client reads a path containing the block from the server. She remaps the position of each block to a random position in the path. For writes, she also updates the value of data in the block and add values from the stash to the path. She then writes the path back to the tree.

3 Privacy Preserving Range Query using PIR and ORAM

We introduce two new approaches for privacy preserving range queries. First approach is an implementation of PIR protocol on an existing range query scheme, [5]. In the second one, we apply Path ORAM method for privacy preserving range queries.

3.1 CPIR for privacy preserving range queries

Setup: In the setup phase, the data is partitioned into buckets with greedy multi-partitioning [5]. The total number of buckets and size of each bucket is set as a power of 2 to utilize a tree structure in the BddCPIR model. The bucket sizes are equal; some dummy values are inserted into buckets if necessary.

Once the buckets are generated, the next step is to send them to the server; here we assume that the data items in buckets are encrypted, therefore, data confidentiality is guaranteed. The server stores the buckets within tree leaves such that a leaf node corresponds to one part of each bucket. If the bucket size is equal to DJ block size, i.e., if a bucket’s items can be stored within a single file, there is only one tree. When buckets are larger, multiple (structurally equivalent) trees are used to store the buckets in a way that the leaves with the same binary representation (location) in each tree is a part of the same bucket. Hence the number of trees is proportional to the bucket size and the size of the tree(s) is determined by the number of buckets.

Query: To perform a range query, the client finds the bucket(s) that have the items within the requested data range using a query translation operation. Based on these bucket ids, she prepares the selection bits to retrieve the related content from the server. Although different bits are required for multiple buckets, for a single bucket, she does not need to compute a different set of selection bits for each tree, since, for a given selection-bit set, the corresponding nodes in the trees map to the same bucket.

Response: Based on the selection bits sent by the client, the server performs BddCPIR on each tree to retrieve the corresponding bucket. Since the performance of the PIR scheme is crucial for efficiently retrieving the buckets, instead of Lipmaa’s BddCPIR scheme, an enhanced version of it, e.g., [9], is used in our implementation. In the new CPIR scheme, the performance of the PIR operations are enhanced by the following changes:
Instead of the binary trees, octal and hexadecimal trees are used to reduce the depth of the tree which yields a significant improvement on the performance. A shared-memory non-trivial parallel algorithm for CPIR operations is introduced to improve the performance further.

Fig. 4. A data distribution example for 2 (binary) trees and 8 nodes.

**Parallel CPIR:** To utilize the parallelism in the best way in the new CPIR scheme, we used a (sub)tree-based data distribution and an adaptive query processing algorithm. When there are more trees than the number of server nodes, the trees are equally distributed to the nodes. Otherwise, when there are more than one node per tree, a tree’s subtrees are equally distributed to the corresponding nodes. Here, the number of nodes per tree is used to decide the height of the subtrees. An example distribution with 2 trees and 8 server nodes is given in Figure 4. With 4 nodes, the subtree roots would be in the first level.

After each (sub)tree is processed, if it is possible the results are combined on a different node to achieve a better parallelism. For example, to combine the results in the first level, in Figure 4, the \((k + 1)^{th}\) node sends its data to the \(k^{th}\) node for \(k = 1, 3, 5, 7\). To combine the results in the root level, the \((k + 2)^{th}\) node sends its data to the \(k^{th}\) node for \(k = 1, 5\). That is for each subtree division, a master node is selected to combine the partial results due to this division, to utilize the nodes as much as possible and obtain a scalable solution.

Within a single node, we use a hybrid, coarse/fine-grain query processing algorithm which adapts itself to the number of trees and cores; when the number of trees per node exceeds the number of available cores in the node, coarse-grain parallelism is employed on the tree level, i.e., each (sub)tree is processed in parallel, but the CPIR operation within a (sub)tree is performed serially. On the other hand, when the number of (sub)trees is less than the number of cores, the cores are distributed to the (sub)trees and each CPIR operation within a (sub)tree is performed in a fine-grain fashion in addition to the coarse-grain parallelism obtained in the (sub)tree level.

**3.2 Path ORAM for privacy preserving range queries**

Implementation of Path ORAM for privacy preserving range queries is rather straightforward compared to the CPIR model. The method does not require any change on the server side. The server is only responsible for sending the path that contains the requested data and writing the path back to the tree without any additional computations. Similarly, the client side operations do not require any fundamental changes.
In the setup phase of the method, the binary tree structure on the server is filled with dummy values. To place the data items into binary tree, the client performs repetitive write operations using the access protocol. The properties of a data item are same with CPIR method. That is each item has several attributes and based on the total size of the item, it is encrypted with AES using a suitable block size. After each retrieval operation, the data is re-encrypted. To enable different ciphertext values for the same data item, some random value is padded to a plaintext for each encryption operation.

For range queries, an extra query processing operation is employed in addition to the original scheme [7]: when a client wants to search for a range, the query processor finds the buckets which store the requested items. Since the query range might be mapped to several buckets, the client may need to perform more than one read operation to retrieve the items.

4 Analysis of Security in Privacy Preserving Range Queries

The proposed approaches assume an honest but curious server model where client is the owner of the database. As mentioned in Section 1, a privacy preserving range query scheme needs to ensure three security conditions, data, query and access pattern confidentiality. Data confidentiality is provided by encrypted storage of data under AES encryption. Query confidentiality is handled by sending bucket ids in CPIR based method and the id of sink node in Path ORAM based method instead of original query range. Finally, leakage of access patterns is prevented by utilizing the security definitions of CPIR and Path ORAM. In the rest of this section these security definitions are briefly explained.

4.1 Security Analysis of CPIR

According to Lipmaa [6], a CPIR protocol achieves client security when it is difficult for a probabilistic polynomial time server to distinguish between two queries \( Q(x_0) \) and \( Q(x_1) \) where \( x_0 \) and \( x_1 \) are client’s selection bits.

Accordingly, to reveal the access pattern, a curious server needs to differentiate the value of selection bits for the target data. In Lipmaa’s CPIR setting, DJ cryptosystem, which is based on Decisional Composite Residuosity assumption, is utilized for cryptographic operations. The cryptosystem satisfies ciphertext indistinguishability by operating randomized algorithms. Thus, differentiating an encrypted selection bit from a random bit string and determining its value is difficult for the server.

4.2 Security Analysis of Path ORAM

Stefanov [7] defines security of Path ORAM based on the indistinguishability of access patterns \( A(y) \) and \( A(z) \) of any two data request vectors \( y \) and \( z \) of same length, considering the failure of scheme with a negligible probability.

The security can be proved by observing the indistinguishability property on position of data and on encrypted data path. Assume the position of a data item, which is stored in private stash, is discovered by server. However, since in
each access the data item is mapped to a new random position and since the new position and former position are statistically independent from each other, server cannot differentiate addresses \([7]\). On the other hand, in each access operation the data path is re-encrypted and encryption is randomized by padding. Thus, encrypted data paths become computationally indistinguishable from a random bit sequence for curious server.

5 A Quantitative Analysis of Path ORAM and CPIR

The volume of exchanged data is an important issue for the efficiency of the range-query algorithms. Hence, a detailed inspection of bandwidth requirements is required for a fair comparison of the schemes.

5.1 Communication complexity analysis

**CPIR:** There are two data transfer phases; the first one is sending the encrypted selection bits to the server. The size of a single selection bit on the \(s^{th}\) level of the tree is \((s + 1)|N|\) where \(|N|\) is the size of the modulus \(N\) \([9]\). Hence, at the lowest level, the size of one encrypted selection bit is \(2|N|\). At the root level, the bit size is \((\log_k n + 1) |N|\) where \(n\) is the number of items in the database and \(k\) is the branching factor of the tree which is 8 and 16 (i.e., octal and hexadecimal) in our implementation. The proposed model requires 7 selection bits in octal trees and 15 selection bits in hexadecimal trees for each tree level. Thus, for one bucket request, we can find the cost of the client-to-server communication in terms of the number of bits as \((k - 1) \times (2 + 3 + \cdots + (\log_k n + 1)) \times |N|\), where \(k = 8\) and \(k = 16\) for octal and hexadecimal trees, respectively. Although there can be multiple trees, sending the selection bits only for a single tree and employing them on all the trees is sufficient.

The second data transfer is the response of the server to the client. For a single bucket request and with a single tree, the number of bits transferred in this phase is \((\log_k n + 1)|N|\). Unlike the client-to-server communication, the volume of the server-to-client communication for CPIR is determined by the number of trees, since in each tree there are data items for the requested bucket. Therefore, the cost needs to be multiplied by the number of trees to compute the total bandwidth usage.

**Path ORAM:** To read the path of the requested bucket from the database, the server sends \(Z \log T\) blocks to the client where \(Z\) is the number of blocks within each bucket, \(T\) is the number of buckets and \(\log T\) is the length of the path containing the retrieved node. And to write the accessed path back to tree, the client sends \(Z \log T\) blocks of data back to the server. Therefore, the total number of blocks sent/received for Path ORAM is \(2Z \log T\). In our experiments, \(Z\) is fixed to 4 as in the original Path ORAM method \([7]\), and each data item is considered as a tuple with 5 integer attributes (including the primary key), i.e., 160 bits. However, the data needs to be stored in encrypted form; thus, a 160 bit plaintext value is mapped to a 256-bit ciphertext block of AES.

The bandwidth usage in case of a single bucket request for the CPIR- and Path ORAM-based range query schemes are presented on Figure 5. The \(y\)-axis
Number of Data items

- 16384
- 131072
- 1048576

Ratio of exchanged bits to database size

- $10^{-3}$
- $10^{-2}$
- $10^{-1}$

255 buckets - ORAM
511 buckets - ORAM
4095 buckets - ORAM
256 buckets - hexCPIR
4096 buckets - hexCPIR
512 buckets - octoCPIR
4096 buckets - octoCPIR

Fig. 5. The ratio of exchanged bits to the database size for the CPIR- and Path ORAM-based range query schemes with different database sizes.

In the figure shows the ratio of the number of exchanged bits to database size (in log scale), whereas the $x$-axis shows the number of data items in the database. As the figure shows, with the same number of buckets, the CPIR-based approach is superior in terms of bandwidth usage and Path ORAM-based scheme consumes much more bandwidth to get a single bucket especially for large databases which is the case for many applications today.

200 buckets - ORAM
511 buckets - ORAM
4095 buckets - ORAM
238 buckets - hexCPIR
4956 buckets - hexCPIR
512 buckets - octoCPIR
4096 buckets - octoCPIR

Fig. 6. The bandwidth usage for the CPIR- and Path ORAM-based schemes to retrieve multiple buckets (x axis) for 1,048,576 items distributed into 4,096 buckets.

Although the above analysis assumes single bucket retrieval, querying for a range may require to retrieve more than one. Figure 6 analyzes the bandwidth usage for (octal and hexadecimal tree based) CPIR- and Path ORAM-based schemes when the number of the retrieved buckets is increasing. As before, the bandwidth usage is given as the ratio of exchanged bits to database size. The analysis is performed for 1,048,576 items which are distributed on 4,096 buckets. As the results show, the path ORAM-based scheme consumes much more bandwidth and the difference increases with the number of buckets retrieved.

Range queries with multiple clients: The additional storage and private stash requirements on the client side make Path ORAM less flexible and ineffi-
cient for many scenarios. The existence of a private stash brings difficulties in a multi-client scenario; when a client performs a read or write operation, she needs to write back the retrieved items to a new path in the server. In addition, she needs to inform the others about these new locations. This can cause a significant amount of computational overhead and bandwidth consumption for each additional client using the database. Thus, the scalability of such a system is questionable. On the other hand, for a multi-client CPIR-based implementation, individual client accesses are seamless, and hence, the scheme is not affected by the number of clients.

To visualize the bandwidth usage in multi-client scenario, a simulation is conducted for 100,000 (single bucket) queries using several number of clients as shown in Figure 7. In the simulation, two different approaches are applied for Path ORAM to update the locations in client’s local. The first one is a push-based approach which requires to push the updated locations to the other clients in each query operation. The second approach is a pull-based approach in which a client needs to perform a comparison with other clients for the most recent value of the position information. Figure 7 shows that CPIR-based method is superior in communication for both approaches, since it does not require any additional bandwidth usage in a multi-client scenario. Additionally, a pull-based Path ORAM method provides better results than a push-based method due to the possibility of communicating with fewer number of clients in each query operation.

![Fig. 7. The bandwidth usage in multiple client scenario for CPIR, pull-based Path ORAM and push-based Path ORAM method](image)

5.2 Computational complexity analysis
Path ORAM is an efficient method for retrieving encrypted data; the server only returns the requested path to the client and does nothing else. Therefore, there is no computational burden on the server. On the other hand, for a single data access, the client side gets $O(\log K)$ blocks, where $K$ is the number of total blocks outsourced to the server. The client needs to decrypt, shuffle, and re-encrypt these blocks for each data access.
Lipmaa’s CPIR requires $O(n)$ computation on the server and $O(\log^2 n)$ computation on the client. Furthermore, since each bucket may be partitioned into several trees, the server’s cost may increase by a constant factor. Hence, a CPIR-based range query scheme is slower than a Path ORAM-based one. However, as we will show in the next section, CPIR can still be a very practical approach for privacy preserving range queries with a tuned tree structure and parallelization.

6 Experiments
We implement the CPIR- and Path ORAM-based privacy preserving range query schemes in C++ using gcc 4.9.2. GNU Multiple Precision Arithmetic Library for large integer arithmetic. OpenMP is used for parallelizing the client- and server-side operations and MPI is employed for server operations in the distributed setting. The single node experiments are performed at a machine running on 64 bit CentOS 6.5 with two Intel Xeon E7-4870 v2 clocked at 2.30 GHz each having 15 cores. The distributed, multi-node experiments are conducted on a cluster with 32 computational nodes connected with 20 Gbps InfiniBand, each with dual Intel Xeon E5520 Quad-core CPUs (with 8MB of L3 cache per processor), 48 GB of main memory, and 64 bit CentOS 6.

To store a data item, we encrypt it by using AES with 256-bit block size. For DJ cryptosystem, 1024-bit modulus is used to provide 80-bit security. The DJ library we used performs encryption/decryption operations sequentially which is the case for many implementations in practice.

Following the bucketization method of [5], the Lineitem table of TPCH benchmark is used [8] as the dataset which is widely used to evaluate database management systems. The table contains more than 6 million data entries, random subsets are created for experiments on smaller datasets. The size of data sets varies from 128 to 16,384 entries. To evaluate multi-dimensional range queries, four integer attributes of Lineitem table – Quantity, Linenumber, ExtendedPrice and Tax – are selected with primary key PartKey-SuppKey.

6.1 Single-node experiments
The first experiment is conducted to compare the client-side performance of our CPIR- and Path ORAM-based implementations. For the CPIR-based scheme, we measure the query times of single bucket retrieval for octal and hexadecimal trees and present them in Table 1 and Table 2, respectively. In the tables, $n$ is the number of data items distributed among a number of buckets which is given in the second column. For the encryption stages, the tree structure and the number of buckets are important parameters to understand the speedup results. As described in Section 5.1, for the octal case, 7 selection bits are encrypted for each tree level except the one containing the root and there are 2, 3, 4, and 5 levels (including the root level) for 8, 64, 512, and 4096 buckets, respectively. For the hexadecimal case, there are 2, 3, and 4 levels for 16, 256, and 4096 buckets, respectively, and the client encrypts 15 bits per level. As the tables show, for the encryption stage with $n = 16,384$ items, we obtained 7.5–8.3 speedup for the octal case and 14.8–16 speedup for the hexadecimal case with 16 threads.
Since the encryption complexity of DJ is quadratic with respect to the input size, which increases as we move to the upper levels in the tree, the encryption tasks for different tree levels do not have the same computational complexity. We aim to distribute a level’s tasks to the threads as evenly as possible to have a better load balance; starting from the most expensive level (the first one), we order the bit encryption tasks according to their levels and assign one task to a thread at a time by tuning the OpenMP scheduling policy. This approach works well for the hexadecimal case for which 15 encryption tasks exist for each level. On the other hand, as Table 1 shows, the speedups for the octal case is not satisfactory with 16 threads since there are only 7 bits to be encrypted at each level and most of the threads cannot get a task from the most expensive level. However with 8 threads, the speedup values for the octal case are between 6.9–7.5 which shows that the load balancing scheme works well as expected when there is enough number of tasks per thread. In the future work, we are planning to parallelize each encryption task and use a hybrid load-balancing approach to have better speedups for the octal case.

For the decryption stage, the main limitation on our parallelization strategy is the number of trees used in the CPIR-based scheme. For each tree, the server returns a ciphertext to the client and we performed the decryption operations on the ciphertexts independently with a single thread per decryption. Hence, when the number of threads exceeds the number of ciphertexts, some cores remain idle. This is why the decryption time does not decrease for some cases when the number of threads increases. On the other hand, we obtain linear scaling for all the cases which is expected since the decryption tasks use the same DJ modulus and have the same complexity. For example, for the octal case (Table 1) with \( n = 16,384 \) items and 512 buckets, the scheme puts 32 items to each bucket. Considering that each 256-bit block can store only 4 data items, the CPIR-based scheme uses 8 trees. For this case, we obtain 8.1 speedup both with 8 and 16 threads.

Overall, as the table shows, the hexadecimal implementation is advantageous for large datasets. The better performance of hexadecimal tree is a result of the less number of levels and hence less complexity of the encryption/decryption operations. For example, for 16,384 items and 4,096 buckets, the octal and hexadecimal tree implementations require a tree with 5 and 4 levels, respectively. Hence, the client-side encrypts 28 and 45 selection bits, respectively. Although the number of bits is more for the hexadecimal case, the costs of client-side encryptions are similar (512 and 487 milliseconds). Furthermore, the load will be better distributed to the threads in our simple hexadecimal implementation since the variance between the task sizes is much less. As a result, with 16 threads, the encryption operations cost 2 times more for the octal case than the hexadecimal case (62 and 32 milliseconds). For the client-side decryption with the same number of data items and buckets, both implementations use a single tree, i.e., a single ciphertext is returned, but the cost in the octal case is more due to a taller tree (25 and 17 milliseconds).

To make the Path ORAM-based operations comparable with those of the CPIR-based scheme, we use the same bucket sizes in our analysis. The average
processing time of the client-side computations in the Path ORAM-based scheme is usually around 1ms for 128, 1024, and 16384 items. Only for 16,384 items and 7-15 buckets, the client spends 6ms for the query preparation. The AES encryption and decryption operations comprise the majority of computation in Path ORAM, whereas the exponentiations performed during the encryption of the selection bits and the decryption of query responses are the main burden on the CPIR-based scheme. As our comparison shows, the Path ORAM-based scheme performs better than the CPIR-based scheme. However, both schemes can be considered as practical considering the core numbers in today's CPUs.

We also measure the cost of the server-side operations for the CPIR-based scheme (Path ORAM-based scheme does not require any server computation). Table 3 shows the server-side cost to retrieve one bucket with octal and hexadecimal trees, respectively. When multiple CPIR trees are employed, the server can process the query on these trees independently. Hence, when the number of threads is smaller than the number of trees, a coarse-grain parallelization can be applied on the tree level. For such cases, the system scales linearly; for example, with an octal tree structure, $n = 16,384$ data items and 512 buckets, the server uses 8 trees as explained above. As Table 3 shows, with 8 threads, i.e., one tree per thread, our implementation obtains 7.9 speedup (the server response time reduces to 748ms from 5,940ms). On the other hand, when the number of threads exceeds the number of trees, the parallelization is not straightforward; one can simply apply a fine-grain parallelism and use all the threads to process a single tree in parallel and repeat the same for all the trees. However, since there exist relatively less number of expensive tasks in the upper levels of a tree, the tree structure can limit the scalability of the fine-grain approach. To alleviate this, we
applied a hybrid scheme where all the trees are processed at the same time and in parallel with the same number of threads: for the octal setting with 16,384 data items, 512 buckets, and 16 threads, 2 threads are assigned to each tree. With this hybrid strategy, we obtained 14.6 speedup (407 ms) with 16 threads, where a pure fine-grain strategy per tree only yields a 6 speedup (991 ms).

<table>
<thead>
<tr>
<th>n</th>
<th>No. buckets</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8</td>
<td>28</td>
<td>17</td>
<td>14</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>85</td>
<td>56</td>
<td>42</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>1,024</td>
<td>64</td>
<td>338</td>
<td>180</td>
<td>85</td>
<td>53</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>743</td>
<td>405</td>
<td>236</td>
<td>152</td>
<td>124</td>
</tr>
<tr>
<td>16,384</td>
<td>8</td>
<td>641</td>
<td>1,827</td>
<td>914</td>
<td>454</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>410</td>
<td>2,712</td>
<td>1,558</td>
<td>680</td>
<td>341</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>5,940</td>
<td>2,976</td>
<td>1,488</td>
<td>748</td>
<td>407</td>
</tr>
<tr>
<td></td>
<td>4,096</td>
<td>5,940</td>
<td>2,976</td>
<td>1,488</td>
<td>748</td>
<td>407</td>
</tr>
</tbody>
</table>

Table 3. Server-side timings for the CPIR-based method (in ms)

<table>
<thead>
<tr>
<th>n</th>
<th>No. buckets</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8</td>
<td>28</td>
<td>17</td>
<td>14</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>85</td>
<td>56</td>
<td>42</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>1,024</td>
<td>64</td>
<td>338</td>
<td>180</td>
<td>85</td>
<td>53</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>743</td>
<td>405</td>
<td>236</td>
<td>152</td>
<td>124</td>
</tr>
<tr>
<td>16,384</td>
<td>8</td>
<td>641</td>
<td>1,827</td>
<td>914</td>
<td>454</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>410</td>
<td>2,712</td>
<td>1,558</td>
<td>680</td>
<td>341</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>5,940</td>
<td>2,976</td>
<td>1,488</td>
<td>748</td>
<td>407</td>
</tr>
<tr>
<td></td>
<td>4,096</td>
<td>5,940</td>
<td>2,976</td>
<td>1,488</td>
<td>748</td>
<td>407</td>
</tr>
</tbody>
</table>

Fig. 8. The number of data items the server-side can provide in a second (DiPS) with various parameters and different number of cores

To analyze the octal and hexadecimal case more clearly, we measured the number of data items the client-side can request and the server-side can provide in a second (DiPS) with various parameters and different number of cores. Figure 8 shows the DiPS values for the server side; a small number of large buckets requires less computation per data item, and a hexadecimal tree structure is better than an octal structure when the same number of buckets have been employed. Thanks to parallelization, with large buckets which are preferable by the queries with large result sets, the server can provide more than 700 data items per second with 16 threads. On the other hand, for the same DiPS value, the client only needs 2 cores (we omit the chart due to space limitations). Hence, as in practice, the client-side requires less computation power compared to the server-side to use the proposed CPIR-based range query scheme at its limit.
6.2 Multi-node experiments

For multi-node experiments, we used a database of size $n = 16,777,216$ and a hexadecimal tree with depth 5. Figure 9 shows the query processing times and the speedups for this experiment with various number of server nodes. Similar to the single-node experiments, a leaf in the tree can store 4 data items. Since there are 16 items per bucket, there are 4 trees in total. Hence, up to 4 server nodes, simply partitioning the trees to the nodes is enough for a balanced load distribution. To use more server nodes, the trees are decomposed into their subtrees; a single division generates 16 subtrees since we are using hexadecimal trees. For each division, a master node is selected to combine the partial results for the corresponding subtree division. The final results for each tree are then sent to the server’s entry node which is also responsible for receiving the query from the client and distributing it to the other processing nodes. The overall communication in the server is negligible; with 32 server nodes, the intra-server communication is around 1% of the overall query response time. The query response time is reduced from 615 secs to 20.5 secs yielding a 30x speedup over a single node (8 core) execution.

7 Conclusion

We proposed two methods for privacy preserving range queries using PIR and ORAM techniques. While most of the existing privacy preserving range query schemes do not deal with hiding query access patterns, our methods aim to prevent the disclosure of access patterns in addition to provide data and query confidentiality. For Private Information Retrieval, we adopted an improved version of Lipmaa’s BddCPIR and applied it on an existing range query scheme [5]. For ORAM, we adapted Stefanov et al.’s [7] Path ORAM. Our analyses show that the Path ORAM-based scheme is much more efficient than the CPIR-based one in terms of computation. However, it is not as flexible as the CPIR-based scheme, and considering its high bandwidth usage, the CPIR-based scheme can be more suitable in practice for various cases such as one with multiple clients.
and slow communication. Furthermore, the computation cost of the CPIR-based scheme can be reduced with parallelization on a distributed- and shared-memory server and client, respectively, which is a very common setting in practice.

Acknowledgments
Erkay Savaş was supported by TÜBİTAK under Grant Number 113E537. Gamze Tillem was supported by TÜBİTAK under BİDEB 2211 program. Kamer Kaya was supported by TÜBİTAK BİDEB 2232 program under grant number 115C018. The authors would like to thank Cengiz Orençik for his valuable comments on the paper.

References
Optimizing MPC for robust and scalable integer and floating-point arithmetic

Liisi Kerik\textsuperscript{1}, Peeter Laud\textsuperscript{1}, and Jaak Randmets\textsuperscript{1,2}

\textsuperscript{1} Cybernetica AS, Tartu, Estonia
\textsuperscript{2} University of Tartu, Tartu, Estonia
{liisi.kerik, peeter.laud, jaak.randmets}@cyber.ee

Abstract. Secure multiparty computation (SMC) is a rapidly matur- ing field, but its number of practical applications so far has been small. Most existing applications have been run on small data volumes with the exception of a recent study processing tens of millions of education and tax records. For practical usability, SMC frameworks must be able to work with large collections of data and perform reliably under such conditions. In this work we demonstrate that with the help of our recently developed tools and some optimizations, the Sharemind secure computation framework is capable of executing tens of millions integer operations or hundreds of thousands floating-point operations per second. We also demonstrate robustness in handling a billion integer inputs and a million floating-point inputs in parallel. Such capabilities are absolutely necessary for real world deployments.

Keywords: Secure Multiparty Computation, Floating-point operations, Protocol design

1 Introduction

Secure multiparty computation (SMC)\textsuperscript{17} allows a group of mutually distrust- ing entities to perform computations on data private to various members of the group, without others learning anything about that data or about the intermediate values in the computation. Theory-wise, the field is quite mature; there exist several techniques to achieve privacy and correctness of any computation\textsuperscript{17, 26, 13, 14, 19}, and the asymptotic overheads of these techniques are known. In practical terms, the search for best implementations and deployment strategies for performing computations on real-world scale is still ongoing. There exist several SMC platforms\textsuperscript{2, 15, 7, 11, 18, 29, 33, 27} and independent implementations of SMC protocols for complex computational tasks\textsuperscript{9, 23} looking for the right trade-offs.

Sharemind\textsuperscript{7, 8} is one of the most mature SMC platforms and the base of some of the largest SMC deployments until now. With the help of Sharemind, we have performed statistical analyses over tens of millions of records\textsuperscript{20, Chap. 6}, and searched for anomalies in a set of 100 million records\textsuperscript{3}. Sharemind achieves the versatility and scalability through a simple security model...
(enabling efficient protocols) and a large set of composable protocols for primitive operations, which can be used as building blocks for large applications. The total number of implemented primitive protocols for integer, fixed- and floating-point operations for arguments of various sizes is significantly over 100. While historically the protocols have been implemented in C++, with more complex protocols invoking simpler ones in hierarchic manner, recently we have introduced a domain-specific language (the Protocol DSL) for specifying them [25].

The Protocol DSL brings at least two benefits. First, it allows tighter composition of protocols, enabling subprotocols with data dependencies to run in parallel without any additional effort from the developer of the protocol set. Second, it allows the developer to try out different implementation options for complex protocols with an effort that is orders of magnitude smaller compared to using C++.

In this paper, we report on our optimizations for the protocols in SHAREMIND’s protocol set, enabled by the Protocol DSL. Many of the improved protocols are used for operations on private floating-point numbers. Our reported optimizations may be useful for other SMC platforms and protocol sets providing private floating-point numbers, as several of our optimizations are not that dependent on particular details of SHAREMIND. In addition to optimizations of private floating-point operations, we also show how the protocol construction toolchain, central to which is the Protocol DSL, allowed us to implement a major architectural change of all protocols with relatively little effort. This provides additional validation of the choices made in [25].

This paper has the following structure. In Sec. 2 we give an overview of SHAREMIND and the protocols it uses, as well as the related work on privacy-preserving floating-point operations. In Sec. 3 we describe our improvements to various floating-point protocols, both generic changes and modifications of specific protocols, as well as the constructions of protocols for new operations. In Sec. 4 we describe another optimization that applies to all protocols in the main protocol set of SHAREMIND. We show that the optimizations in this and previous section improve the performance of protocols for various operations. In Sec. 5 we give a more thorough description on how we have measured the performance of the protocols of SHAREMIND. We provide precise running times of certain protocols, thereby making clear the current state of the art. Finally, we conclude in Sec. 6.

2 Background

In a SHAREMIND deployment, the involved parties are divided into three classes which may overlap: the input parties provide inputs to the private computation, the computation parties execute the SMC protocols for performing operations with private data, and result parties learn the result(s) of the computation [4]. While the architecture of SHAREMIND supports the use of several SMC protocol sets [6], the main set in use is based on additively sharing the private values among three computing parties [8]. The sharing can be over any finite
ring and there are protocols to convert between different rings. Hence the input parties secret share their inputs among computation parties, and the result parties recombine the shares of outputs they receive from computation parties. The computation parties follow the description of the private functionality specified in the Secrec language [6], invoking the SMC protocols in specified order.

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Sharemind’s protocol set provides security against one passively corrupted party. Its security and privacy guarantees are composable, allowing the security of complex protocols to be deduced from the security of its component protocols [5]. The development of secure protocols is also greatly assisted by a protocol privacy checker [30] for the Protocol DSL [25].

Typically, rings $\mathbb{Z}_{2^n}$ are used in Sharemind applications and supported by the Protocol DSL. In the following, we let $[x]$ denote the value $x$ which has been secret-shared among the computing parties, and $[x]_i$ denotes the $i$-th party’s share.

For private numeric computations (e.g. for the satellite collision analysis [21]), Sharemind features a set of protocols for working with secret-shared fixed-point and floating-point numbers [21, 24]. In this protocol set, a floating-point number $x$ is represented as $x = (-1)^s \cdot f \cdot 2^e$, where $s \in \{0, 1\}$ is the sign bit, $f \in \mathbb{Z}_{2^m}$ the significand, and $e \in \mathbb{Z}_{2^n}$ the exponent. The representation with $(m, n) = (32, 8)$ (resp. $(m, n) = (64, 11)$) is called single precision (resp. double precision). For a private value, each part is separately secret-shared among the computing parties. The same representation (plus an indication whether the number is 0) is used also by Aliasgari et. al [1] who have built a private floating-point protocol set implementing arithmetic operations and a number of elementary functions on top of Shamir’s threshold secret sharing [32]. In a different line of work, protocols for private floating-point operations have been built atop garbled circuits or the GMW protocol set [28, 16] with various optimizations.

Internally, many of our floating-point protocols call protocols for computations on private fixed-point numbers. In our protocols, a fixed-point number $x$ is represented as an integer $x \cdot 2^M$ for a suitable $M$. Several sets of SMC protocols for fixed-point computations (including both arithmetic operations and elementary functions) have been proposed [12, 24]. Our Protocol DSL has allowed us to experiment with the details of these protocols and propose more efficient implementations.

3 Improvements in protocol design

In our floating-point protocols, we use the following operations as primitive building blocks:

- Zero-extension of secret shared integers denoted with $\text{Extend}(u, n)$ where $[u] \in \mathbb{Z}_{2^m}$. This operation converts a private integer from $\mathbb{Z}_{2^m}$ to $\mathbb{Z}_{2^{m+n}}$ without changing its value.
- Dropping some least-significant bits of a secret shared integer, denoted with $\text{Cut}(u, n)$ where $[u] \in \mathbb{Z}_{2^m}$ and $n \leq m$. The cut operation removes $n$
Algorithm 1: Protocol PowArr for integer powers of a fixed-point number.

Data: $\langle x \rangle, k, n, n'$

Result: Computes the powers of a secret fixed-point number. Takes in a secret fixed-point number $\langle x \rangle$ with 0 bits before and $n$ bits after the radix point. Outputs a secret fixed-point array $\{\langle x^i \rangle\}_{i=1}^k$ with $n' + n$ bits before and $n$ bits after the radix point.

1. if $k = 0$ then
2.   return $\{\}$
3. else
4.   $l \leftarrow \lceil \log_2 k \rceil$
5.   $\langle x^1 \rangle \leftarrow \text{Extend}(\langle x \rangle, n' + (l + 1)n)$
6.   for $i \leftarrow 0$ to $l - 1$ do
7.     $\{\langle x^j \rangle\}_{j=2^i+1}^{2^{i+1}} \leftarrow \text{MultArr}(\langle x \rangle, \{\langle x^j \rangle\}_{j=1}^{2^i})$
8.     for $j \leftarrow 1$ to $2^{i+1}$ do in parallel
9.       $\langle x^j \rangle \leftarrow \text{Cut}(\langle x^j \rangle, n)$
10. end
11. end
12. return $\{\langle x^i \rangle\}_{i=1}^k$
13. end

least significant bits of $\langle u \rangle$ and results in an $(m - n)$-bit integer. It computes $\lfloor u/2^n \rfloor$ more efficiently than division or shift-right operation.

- Multiplication of integer with an array of integers $\text{MultArr}(\langle u \rangle, \{\langle v_i \rangle\}_{i=1}^k)$, where $\langle u \rangle \in \mathbb{Z}_{2^n}$ and $\langle v_i \rangle \in \mathbb{Z}_{2^n}$ for every $i \in \{1, \ldots, k\}$. The operation results in an array $\{\langle w_i \rangle\}_{i=1}^k \in \mathbb{Z}_{2^n}$, where $w_i = u \cdot v_i$. The implementation is straightforward based on regular integer multiplication protocol. Efficiency is improved by sending the shares of $u$ only once instead of $k$ times.

We do not describe the implementations of those operations here. However, all of them are relatively straightforward to implement using the tools provided in [8].

3.1 Efficient polynomial evaluation

Most of our floating-point functions are implemented using polynomial approximation. For example, when computing the square root of $2^e \cdot f$ we approximate the square root of fixed-point $f$ with a polynomial and return $2^{e/2} \cdot \sqrt{f}$ [24, Alg. 5]. Fast and precise fixed-point polynomial evaluation is important to ensure the speed and accuracy of floating-point operations. Recall that fixed-point addition is just regular integer addition. Multiplication requires extending both inputs to larger integers, integer multiplication and dropping the lowest bits.

We have significantly improved upon the fixed-point polynomial evaluation presented in [24, Alg. 1]. Improved protocol for polynomial evaluation is presented in Alg. 2 and a helper function for evaluating integer powers of a fixed-point number is presented in Alg. 1. First, polynomial coefficients are now represented in two’s complement form as opposed to using sign bits. This means we
Algorithm 2: Fixed-point polynomial evaluation protocol.

Data: \([x], \{c_i\}_{i=0}^k, n, n'\)

Result: Computes a public polynomial on a secret fixed-point number. Takes in a secret fixed-point number \([x]\) with 0 bits before and \(n\) bits after the radix point and public fixed-point coefficients \(\{c_i\}_{i=0}^k\) with \(n' + n\) bits before and \(n\) bits after the radix point (the highest \(n\) bits are empty). Outputs a secret fixed-point number \([y]\) with 0 bits before and \(n\) bits after the radix point that is the value of the polynomial at \(x\).

1. \(\{([x^i])_{i=1}^k\} \leftarrow \text{PowArr}([x], k, n, n')\)
2. \([z_0]\) ← \(\text{Share}(c_0)\)
3. for \(i \leftarrow 1\) to \(k\) do in parallel
4.  \([z_i]\) ← \(c_i \cdot [x^i]\)
5. end
6. for \(i \leftarrow 0\) to \(k\) do in parallel
7.  \([z_i]\) ← \(\text{Trunc}([z_i], n')\)
8. end
9. \([y]\) ← \(\text{Cut}(\text{Sum}(\{[z_i]\}_{i=0}^k), n)\)
10. return \([y]\)

do not need to pick different multiplication results depending on the sign bits. Second, we have improved the efficiency of fixed-point multiplications which are used to evaluate the polynomial. The algorithm in [24] uses ordinary fixed-point multiplications throughout. Fixed-point multiplication requires extending the operands beforehand, multiplying, and then cutting off the lowest bits. This approach is costly, and we would like to avoid extending the operands before each multiplication. So, we extend the argument of the polynomial only once, in the beginning, by a sufficient number of bits to allow for all subsequent cuts. This approach is costly, and we would like to avoid extending the operands before each multiplication. So, we extend the argument of the polynomial only once, in the beginning, by a sufficient number of bits to allow for all subsequent cuts. This approach is analogous to the one used in [25, Alg. 8] for computing the product of several fixed-point numbers. Third, we have made the last round of polynomial evaluation more efficient; while in [24, Alg. 1] the powers of the argument are multiplied by the corresponding coefficients, the lowest bits of the results are cut off, and then they are added up to find the value of the polynomial, we first perform the summation and then cut off the lowest bits of the sum, thus replacing \(k\) cut operations with 1. In addition to efficiency this shortcut slightly improves precision as it results in smaller rounding error of the end result.

Our polynomial evaluation algorithm is in a way less general than [24, Alg. 1] as both the argument and the result have to be in range \([0, 1)\). However, this approach is sufficient for all the floating-point functions that we have implemented. In fact, this striction offers an advantage as it ensures that the powers of \(x\) do not overflow. Note that we do not place any restrictions on the size of the coefficients, while [24, Alg. 1] requires the coefficients to fit into the same fixed-point format as the argument and the result. In [24, Alg. 5], when computing the square root of a fixed-point number in range \([0.5, 1)\), the argument has to be shifted right in order to achieve a fixed-point format with enough bits
before radix point to fit in the coefficients; our approach allows for coefficients that are larger than the argument, and therefore, no precision is lost through shifting out the lowest bits of the argument.

We, similarly to [24], approximate functions by interpolating through Chebychev nodes [10, p. 521]. We have implemented two adjustments which result in better approximations.

First, sometimes we want the result to be in a certain range. For example, we assume that the result of \( 2^x - 1 \) where \( x \in [0, 1) \) ought to be in range \( [0.5, 1) \). However, approximation errors might cause results outside the range and overflows. In [24] this problem was solved by the so-called correction protocol which normalizes the result into the correct range. We get a suitable result directly, with no need for the correction step. If we interpolate function \( f(x) \) in range \( (a, b) \) and we need \( f(a) \) to be rounded upwards and \( f(b) \) to be rounded downwards we pick a small positive constant \( \epsilon \) and interpolate function \( f(x) + \epsilon \cdot (a + b - 2x)/(b - a) \) instead. The small linear term ensures that approximation errors are in the right direction. If we want to round \( f(a) \) downwards and \( f(b) \) upwards then \( \epsilon \) has to be negative. Should need arise to achieve errors in the same direction on both ends a small quadratic term added to the function can achieve this result.

Second, large coefficients pose a problem: due to the particularities of fixed-point polynomial evaluation they can result in large approximation errors and make the algorithm too imprecise for practical use in some cases. For example, interpolating \( \text{erf}(8x) \) in range \( [0.125, 0.25) \) with 17 nodes results in coefficients that are larger than \( 2^{30} \) and therefore need 31 bits before radix point; when evaluating this polynomial, the rounding errors inherent to fixed-point computations result in an extremely imprecise approximation. We can improve the situation by noting that the first three bits of the input are always the same (001) and shifting the input 3 bits to the left, which amounts to multiplying it by 8 and subtracting 1. The initial range \( [0.125, 0.25) \) is mapped into \( [0, 1) \) and the new function that has to be interpolated is \( \text{erf}(x + 1) \). Interpolation with 17 nodes yields coefficients which are less than 1 and therefore require 0 bits before radix point and thus, precision is improved, and in this example the length of most variables in polynomial computation is reduced by almost 4 bytes. This approach of shifting out the known highest bit(s) of the argument and modifying the function for interpolation has improved the efficiency and precision of square root, logarithm, and error function.

As a result of aforementioned changes, evaluating a polynomial of degree 16 on a 64-bit fixed-point number takes 57 rounds and 7.5 KB of communication, while with the old algorithm, it takes 89 rounds and 27 KB of communication.

### 3.2 Additional improvements to floating-point protocols

In addition to improvements made to polynomial evaluation that benefit most floating-point functions, we have also modified other protocols from [24], namely inverse, square root, exponent function, and error function.
The new inverse protocol has been presented in [25, Alg. 8]. We have found that correction of fixed-point inverse approximation results is not necessary as with this method $0.5^{-1}$ is always rounded down and $1^{-1}$ is always rounded up.

Computation of exponent function begins by separating the input $x$ into whole part and fractional part. In [24, Alg. 6] the whole part $\lfloor x \rfloor$ is computed in integer format and converted to floating-point format. The fractional part $\{x\}$ is computed through floating-point subtraction: $\{x\} = x - \lfloor x \rfloor$. Then $\{x\}$ has to be converted to fixed-point format in order to approximate $2^x$. Instead of combining costly integer to floating-point conversion and floating-point subtraction, we have designed a special separation protocol which efficiently separates a floating-point number into whole and fractional part (in integer and fixed-point format, respectively) by obliviously choosing between all possible results.

Another optimization we have devised for exponent function is an improvement to the computation of polynomials on $\{x\}$ and $1 - \{x\}$. Instead of computing the powers of $1 - \{x\}$ in ordinary manner we use the powers of $\{x\}$ and binomial coefficients. This employs only fast, local operations - multiplication by a public integer and addition. (For why we need to compute the value of a polynomial on both $\{x\}$ and $1 - \{x\}$ see [24, Alg. 6].)

When $2^{\{x\}}$ has been found and converted to floating-point format, the end result is computed as $2^{\lfloor x \rfloor} \cdot 2^{\{x\}}$. In [24, Alg. 6] this is achieved through floating-point multiplication. We have found a more efficient approach: since $2^{\{x\}}$ is a floating-point number we can just add $\lfloor x \rfloor$ to the exponent (which allows us to avoid an integer to floating-point conversion and a floating-point multiplication).

Finally, we have added a new feature to exponent function. When $2^x$ becomes so small it cannot be represented accurately, we round the result down to zero.

In [24, Alg. 7] $\text{erf}(x)$ is approximated by $2x/\sqrt{\pi}$ if $x < \epsilon$ and 1 if $x \geq 4$. The interval $[-\epsilon, 4]$ is divided into 4 pieces and in each one the function is approximated with a different polynomial. In our implementation, double-precision $\text{erf}(x)$ is approximated by 1 if $x \geq 8$. The interval $[\epsilon, 8]$ is divided into 8 pieces; in first six the function is approximated with polynomials and in last two with constants. We compute several different polynomials (4 in single-precision case and 6 in double-precision case) on the same number and perform oblivious choices in the end. We can optimise this calculation by computing the powers of the argument only once as they are the same for all polynomials. But the main improvement in performance comes from restructuring the algorithm to compute only the correct value of $\text{erf}(x)$ instead of computing several different values and obliviously choosing between them in the end. In [24, Alg. 7] several possible shift rights of the significand are computed (essentially giving us several possible results of the floating-point to fixed-point conversion). On all of them, error function is computed, and finally, the correct result is picked obliviously. We have reversed the order of the last two steps: first, we obliviously pick the correct shift right of the significand (essentially performing a floating-point to fixed-point conversion) and then we compute the error function on the single correct value.

Our improvements have increased precision compared to [24]. The maximum relative error of inverse is $2.69 \cdot 10^{-9}$ for single precision and $7.10 \cdot 10^{-19}$ for
double precision (compared to $1.3 \cdot 10^{-4}$ and $1.3 \cdot 10^{-8}$ in [24]). For square root our errors are respectively $4.92 \cdot 10^{-9}$ and $1.30 \cdot 10^{-15}$ (compared to $5.1 \cdot 10^{-6}$ and $4.1 \cdot 10^{-11}$ in [24]). In a few cases we have achieved better accuracy guarantees than what IEEE 754 single- and double-precision floating-point numbers allow. This is possible because we are using slightly longer fractional parts.

3.3 New floating-point protocols

In addition to improving the floating-point protocols published in [21, 24, 25] we have also designed a few new ones, namely logarithm, sine, floor and ceiling. Here we shall present a short explanation of logarithm and sine.

In order to compute the binary logarithm of a floating-point number we note that $\log_2(2^e \cdot f) = e + \log_2 f$. As $f$ is in range $[0.5, 1)$ its binary logarithm is in range $[-1, 0)$. However, in order to easily convert it to a floating-point number, we would like to get a result in range $[0.5, 1)$. Therefore, we transform the expression above as follows: $e + \log_2 f = (e - 2) + 2(\log_4 f + 1)$. If $f$ is in range $[0.5, 1)$ then the value of $\log_4 f + 1$ is in range $[0.5, 1)$. This is the function that we approximate with a fixed-point polynomial. For double precision, we split the interval into two equal parts and use two different polynomials. Finally, $e - 2$ is converted to floating-point format and the end result is computed through floating-point addition. Near 1 we use second degree Taylor polynomial $\log_2 x \approx \log_4 e \cdot (x - 1)(3 - x)$ to achieve better precision. In order to convert binary logarithm to natural logarithm we use the conversion $\ln x = \ln 2 \cdot \log_2 x$.

The algorithm for computing the sine is relatively straightforward as we can use to our advantage all kinds of symmetry inherent to the function. First, we divide the argument by $2\pi$ and find the fractional part in fixed-point format, thus reducing the computation to two full turns (from $-2\pi$ to $2\pi$). We note that $\sin(-x) = -\sin x$, $\sin(x + \pi) = -\sin x$, and $\sin(\pi/2 - x) = \sin(\pi/2 + x)$. Thus, we have reduced the computation to one quarter-turn (from 0 to $\pi/2$). Then we use fixed-point polynomial approximation and convert the end result to floating-point format. When the argument is near zero we use the approximation $\sin x \approx x$ to achieve better precision.

4 Optimization techniques

The Protocol DSL has allowed us to easily apply certain optimizations across the entire suite of protocols employed in SHAREMIND. They are described in the following. The optimizations are specific to the “main” protocol set [8] of SHAREMIND based on additive secret sharing over finite rings, using three computing parties.

4.1 Shared random number generators

To ensure that a party’s view in a protocol could be generated from only its inputs, we commonly use the resharing protocol, to ensure independence from
Table 1. Speedups of shared RNG (SRNG) and symmetric multiplication protocols over the regular multiplication. The speedups have been measured from 1 element inputs to $10^8$ element input vectors.

<table>
<thead>
<tr>
<th>Bit-width</th>
<th>SRNG 10^6</th>
<th>SRNG 10^8</th>
<th>Symmetric 10^6</th>
<th>Symmetric 10^8</th>
<th>SRNG &amp; Symmetric 10^6</th>
<th>SRNG &amp; Symmetric 10^8</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.03</td>
<td>1.03</td>
<td>1.48</td>
<td>1.08</td>
<td>1.44</td>
<td>1.08</td>
</tr>
<tr>
<td>32</td>
<td>0.95</td>
<td>0.98</td>
<td>1.34</td>
<td>1.09</td>
<td>1.45</td>
<td>1.02</td>
</tr>
<tr>
<td>16</td>
<td>0.85</td>
<td>0.90</td>
<td>1.14</td>
<td>1.18</td>
<td>1.17</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>0.96</td>
<td>0.96</td>
<td>1.03</td>
<td>1.04</td>
<td>1.11</td>
<td>1.07</td>
</tr>
</tbody>
</table>

other parties’ inputs and outputs. For example, usually every input of a protocol is explicitly reshared. The resharing protocol takes a private value $[u] \in R$ and returns a $[v] \in R$ such that $u = v$ and all shares $[v]_i$ are uniformly distributed and independent of the shares $[u]_j$. The protocol is implemented as follows: each party $P_i$ generates a random value $r_i \in R$ and sends it to the next computing party $P_{i+1}$, adds the generated value $r_i$ to the input share $[u]_i$, and subtracts the random number $r_{i-1}$ received from the previous computing party $P_{i-1}$. The shares of the output $[v]$ of the protocol are $([u]_1 + r_1, [u]_2 + r_2, [u]_3 + r_3)$. We see that $v = [v]_1 = [v]_2 = [v]_3 = u$.

We can spot a common pattern that occurs in resharing (and in some other primitive protocols): a party generates a random number and sends it to some other party. This pattern can be optimized by letting both parties generate the same random number using a common random number generator (RNG). Analysis of our protocols shows that network communication can be reduced by 30% to 60% using this technique (exactly 60% in the case of integer multiplication protocol). This optimization is not new and has previously been used in [22]. Our toolchain around the Protocol DSL allows this optimization to be automatically introduced, with no changes to the specification of the protocols. The optimization itself is straightforward on our intermediate representation: we detect randomness nodes that are sent to one other computing party, and transform them to instead take use of shared randomness nodes.

We have manually implemented this optimization for the multiplication protocol (for which the Protocol DSL has not been used) and compared the performance to the unoptimized version to validate the effectiveness of this modification. Multiplication protocol has been chosen because of its simplicity, efficiency, ubiquity in application, and because it is one of the least computation heavy protocols. The comparison was performed using the methodology described in Sec. 5 and the results are displayed in Table 1. We see a slowdown of at most 15% on small input lengths (up to one hundred elements), but for large inputs we see a universal speedup that reaches up to 60%. The performance of 64-bit multiplication has been universally improved. The slowdown on small inputs can be explained by a slight increase in computation overhead (critical path became longer due to invoking the shared RNG in the end of the protocol) and the
Algorithm 3: Multiplication protocol.

Data: Shared values \( [u], [v] \in R \)

Result: Shared value \([w] \in R \) such that \( uv = w \).

1. \( [u] \leftarrow \text{Reshare}([u]) \)
2. \( [v] \leftarrow \text{Reshare}([v]) \)
3. All parties \( P_i \) perform the following:
   4. Send \( [u]_i \) and \( [v]_i \) to \( P_{n(i)} \)
   5. Receive \( [u]_{p(i)} \) and \( [v]_{p(i)} \) from \( P_{p(i)} \)
   6. \( [w]_i \leftarrow [u]_i \cdot [v]_i + [u]_{p(i)} \cdot [v]_i + [u]_i \cdot [v]_{p(i)} \)
7. \( [w] \leftarrow \text{Reshare}([w]) \)
8. return \([w] \)

Algorithm 4: Symmetric multiplication protocol.

Data: Shared values \( [u], [v] \in R \)

Result: Shared value \([w] \in R \) such that \( uv = w \).

1. \( [u] \leftarrow \text{Reshare}([u]) \)
2. \( [v] \leftarrow \text{Reshare}([v]) \)
3. All parties \( P_i \) perform the following:
   4. Send \( [u]_i \) to \( P_{n(i)} \) and \( [v]_i \) to \( P_{p(i)} \)
   5. Receive \( [u]_{p(i)} \) from \( P_{p(i)} \) and \( [v]_{n(i)} \) from \( P_{n(i)} \)
   6. \( [w]_i \leftarrow [u]_i \cdot [v]_i + [u]_{p(i)} \cdot [v]_i + [u]_{n(i)} \cdot [v]_{p(i)} \)
7. \( [w] \leftarrow \text{Reshare}([w]) \)
8. return \([w] \)

Speedup can be explained by the decrease in network communication. In fact, network communication is reduced by exactly 60%.

4.2 Symmetric protocols

Multiplication protocol in additive schemes is commonly presented as Alg. 3 such as in [8] and [25]. The given protocol is perfectly reasonable when the SRNG optimization is not used: the resharing sub-protocol sends the network messages in one direction and the multiplication protocol itself in the other. As a result the communication channels are under similar workload. However, using the SRNG optimization results in a protocol that sends network messages only over one of the two network channels. We propose a small modification in the form of Alg. 4 as an alternative multiplication protocol that uses the network in a balanced manner. The correctness and security of the algorithm can be shown the same way as it was shown for the multiplication protocol in [8].

The symmetric protocol provides a small performance gain over the SRNG optimized protocol. The comparison against our legacy multiplication protocol (see Table 1) shows better results and disappearance of the slowdown present with only the SRNG optimization. Only the 8-bit multiplication experiences a small slowdown in a few cases. We predict that the speedups will be greater in
Table 2. Speedup of optimized floating-point protocols.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precision</th>
<th>Speedup on given input length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$10^0$</td>
<td>$10^1$</td>
</tr>
<tr>
<td>[x] + [y]</td>
<td>single</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.97</td>
</tr>
<tr>
<td>[x] × [y]</td>
<td>single</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>1.03</td>
</tr>
<tr>
<td>√[x]</td>
<td>single</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>1.06</td>
</tr>
</tbody>
</table>

In a setting where network latency is worse or the available bandwidth is smaller because in these cases the network will become the dominant bottleneck. This claim is supported by the evidence that the speedups improve as the protocols need to send more data over the network (larger bit-widths or larger input vectors).

This modification can be applied to many other protocols, but a few of the protocols are inherently asymmetric (such as squaring a value, or finding the bitwise conjunction of a single bit with a 64-bit integer). For all asymmetric protocols we can implement two versions that are unbalanced in different directions, and pick versions of them such that overall the communication is roughly balanced (we do not expose this facility to the end user). This optimization has been applied manually as the set of primitive protocols is manageable and the protocol DSL enables such changes easily. We have not explored the possibility of automatically performing communication balancing.

4.3 Speedup over previous results

We have applied the systematic optimizations presented in this section to all our protocols and compared the results against operations without those optimizations. In addition to the optimizations mentioned previously we have also eliminated many resharin calls (this optimization does not reduce network communication) as allowed by [5] and verified the security of resulting protocols using our privacy analyser [30]. Table 2 shows comparison results for floating-point addition, multiplication and square root. These protocols provide a rough idea of how the optimizations fare across all protocols.

Table 2 shows an almost universal improvement in performance. In a few cases single-precision floating-point operations perform slightly worse (less than 10%) but only on small input sizes. In the case of inputs of length 100 and more we see significant speedups across the board. In a few cases speedups reach over 80%.
5 Large-scale performance evaluation

Benchmarking was performed on a dedicated cluster of three computers connected with 10Gbps Ethernet. Each computer was equipped with 128GB DDR4 memory, two 8-core Intel Xeon (E5-2640 v3) processors and was running Debian 8.2 Jessie (15th Sep 2015). Both memory overcommit and swap were disabled. During benchmarking only the necessary system processes and some low overhead services (such as SSH and monitoring) were enabled.

A single run-time measurement was computed by taking the running times of each of the computing parties and finding the maximum of those. This is necessary as a protocol may terminate faster for some participants and the maximum reflects the time it takes for the result of the operation to become available to all. The average running time was estimated by computing the mean of all the measurements. On every input length we performed at least 5 repetitions (10 for integer operations) and, to reduce variance, significantly more on small input lengths (up to 10000 repetitions). Measurements were performed in a randomized order because we found that running the tests sequentially in an increasing size of inputs gave significantly better performance results. Sequential order results in a steady increase of network load which is predictable for the networking layer but is not a very realistic scenario for all SMC applications.

Performance results for floating-point operations are presented in Table 3. We have measured addition, multiplication, comparison, reciprocal, square root, exponentiation, natural logarithm, sine, and error function from 1 element input to one million element input vectors. All the results have been presented in operations per millisecond (thousands of operations per second). Looking at the table, it is clear that performance scales very well with vectorization: only a few hundred scalar operations can be executed per second but by computing on many inputs in parallel we can perform hundreds of thousands of operations per second.

We have also measured the performance of integer and fixed-point multiplication operations (Table 4). The fixed-point operations, especially addition and multiplication, have turned out to be useful tools in implementing efficient higher-level applications. As the respective floating-point operations are rather slow, the computations relying heavily on them may become impractical (for example, floating-point addition [21, Alg. 4] requires private shifts which makes it a costly operation). While not a universal solution, efficient signed fixed-point operations alleviate the problem in many cases.

We have also evaluated private integer multiplication to establish a baseline, against which to compare more complex protocols when choosing the operations to be used in a larger application. We have limited the performance evaluation of multiplication to $10^5$ element input vectors. This is due to memory limitations: a single $10^{10}$ element vector of 64-bit integers takes roughly 80 gigabytes of RAM (it would be possible to only allocate a single vector and use that as both input and output, but this would compute square and not product). Capability to handle arrays of $10^6$ elements with ease demonstrates the robustness of our platform.
Table 3. Performance (in operations per millisecond) of optimized floating-point operations. Combing all manual and automatic optimizations presented in this work. Variables $x$ and $y$ denote floating-point numbers.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precision</th>
<th>OP/ms on given input length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10^0</td>
<td>10^1</td>
</tr>
<tr>
<td>$x + y$</td>
<td>single</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.27</td>
</tr>
<tr>
<td>$x \times y$</td>
<td>single</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.54</td>
</tr>
<tr>
<td>$x &lt; y$</td>
<td>single</td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.97</td>
</tr>
<tr>
<td>$x^{-1}$</td>
<td>single</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.23</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>single</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.21</td>
</tr>
<tr>
<td>exp(x)</td>
<td>single</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.16</td>
</tr>
<tr>
<td>ln(x)</td>
<td>single</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.12</td>
</tr>
<tr>
<td>sin(x)</td>
<td>single</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.12</td>
</tr>
<tr>
<td>erf(x)</td>
<td>single</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>0.18</td>
</tr>
</tbody>
</table>

We have compared the performance of arithmetic operations and square root against previous works. Unfortunately it was not possible to provide comparison in an identical setups as both previous works we compare against have the performance measures on a 1 Gbps Ethernet connection over LAN (opposed to our 10 Gbps connection over LAN). However, we found that we never came close to saturating a 1 Gbps of the connection. Performance in [31] was measured on a cluster of three nodes each equipped with 48 GB of RAM and 12-core 3 GHz Intel CPUs supporting AES-NI and HyperThreading. Performance in [16] was measured on two desktop computers each equipped with a 3.5 GHz Intel Core i7 CPU and 16 GB of RAM (the number of cores was unspecified).

In the case of additive 3-party secret sharing the best results so far have been obtained in [31]. In the case of scalar operations our results show 132 fold speedup for addition, 67 fold speedup for multiplication and 618 fold speedup for square root. The speedups also remain good for $10^4$ element input vectors: 16, 14 and 416 fold respectively. Additionally [31] reports the performance of garbled circuit based on IEEE 754 floating-point numbers. Compared to those
Table 4. Performance of optimized integer and signed fixed-point multiplication. Numbers are provided in operations per second with suffix K denoting thousands and M denoting millions.

<table>
<thead>
<tr>
<th>Type</th>
<th>10^0</th>
<th>10^1</th>
<th>10^2</th>
<th>10^3</th>
<th>10^4</th>
<th>10^5</th>
<th>10^6</th>
<th>10^7</th>
<th>10^8</th>
<th>10^9</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8</td>
<td>7.4K</td>
<td>71.6K</td>
<td>703.5K</td>
<td>5.8M</td>
<td>40.1M</td>
<td>28.0M</td>
<td>37.9M</td>
<td>41.5M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uint16</td>
<td>7.0K</td>
<td>68.4K</td>
<td>663.4K</td>
<td>5.4M</td>
<td>34.0M</td>
<td>29.5M</td>
<td>35.0M</td>
<td>37.1M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uint32</td>
<td>6.6K</td>
<td>65.7K</td>
<td>629.7K</td>
<td>5.0M</td>
<td>17.1M</td>
<td>22.4M</td>
<td>18.8M</td>
<td>20.7M</td>
<td>21.4M</td>
<td></td>
</tr>
<tr>
<td>uint64</td>
<td>6.4K</td>
<td>63.5K</td>
<td>586.9K</td>
<td>4.3M</td>
<td>11.2M</td>
<td>12.1M</td>
<td>10.5M</td>
<td>12.1M</td>
<td>13.7M</td>
<td>13.3M</td>
</tr>
<tr>
<td>fix32</td>
<td>640</td>
<td>6.0K</td>
<td>51.2K</td>
<td>435.4K</td>
<td>344.1K</td>
<td>361.1K</td>
<td>369.4K</td>
<td>351.6K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fix64</td>
<td>680</td>
<td>6.2K</td>
<td>46.6K</td>
<td>187.3K</td>
<td>184.3K</td>
<td>186.0K</td>
<td>187.6K</td>
<td>179.0K</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

we provide 13, 20 and 27 fold speedups in the case of scalars and 102, 364 and 495 fold speedups in the case of 10^4 element input vectors.

While garbled circuit approach is not directly comparable to secret sharing we also compare our results against [16] which provides, to our knowledge, as of now, the best performance for 2-party garbled circuit approach. For scalar operations we are, at worst, 80% slower, and in case of 10^4 element input vectors at worst 50% slower, and at best 4.6 times faster. This is considering only online time. When offline time is also taken into account we report similar performance for scalar operations and significant speedups for vectorized ones (over 40 fold). These comparisons are against the better of GMW (vector operations) and Yao (scalar operations).

6 Conclusions

We have demonstrated the current state of the art in the performance of SMC protocols for numeric computations. Our results show that with careful design and the right set of tools, significant performance improvements are still possible. But currently, as Table 4 shows, the performance of SMC operations on modern but reasonably-spec’d hardware is comparable to a computer with a 80386 processor.

References


On-the-fly Homomorphic Batching/Unbatching

Yarkın Doröz1, Gizem S. Çetin1, and Berk Sunar1

Worcester Polytechnic Institute
{yedoroz,gscetin,sunar}@wpi.edu

Abstract. We introduce a homomorphic batching technique that can be used to pack multiple ciphertext messages into one ciphertext for parallel processing. One is able to use the method to batch or unbatch messages homomorphically to further improve the flexibility of encrypted domain evaluations. In particular, we show various approaches to implement Number Theoretic Transform (NTT) homomorphically in Fast Fourier Transform (FFT) speed. Also, we present the limitations that we encounter in application of these methods. We implement homomorphic batching in various settings and present concrete performance figures. Finally, we present an implementation of a homomorphic NTT method in which we process each element in an independent ciphertext. The advantage of this method is we are able to batch independent homomorphic NTT evaluations and achieve better amortized time.

Keywords: Homomorphic encryption, homomorphic batching, homomorphic number theoretic transform.

1 Introduction

Fully Homomorphic Encryption (FHE) is an encryption method that allows to perform arbitrary circuit or function evaluations on encrypted data without the need for decryption of the ciphertexts. The first FHE scheme was a lattice-based construction introduced by Gentry [12] in 2009. In 2010, Gentry and Halevi [15] simplified the construction and completed the first practical FHE implementation. Even with the optimizations the FHE scheme lacked in performance, since a crucial operation called recryption had to be performed after each bit AND operation which was taking 30 seconds. After the first FHE implementation various schemes [13, 9, 4, 5, 14, 3] have emerged with different optimization techniques on fully or somewhat homomorphic encryption (SHE). In [26] batching and SIMD operations were introduced to pack multiple messages into a ciphertext and thereby allow for parallel homomorphic evaluations. Other operations such as bootstrapping [12], relinearization [23], modulus reduction [5, 3], key switching [3] and flattening [17] are used as key and noise management techniques permitting the evaluation of deeper circuits with similar parameter sizes.

In [3] Brakerski, Gentry and Vaikuntanathan implemented a leveled FHE scheme that is capable of evaluating polynomial-size circuits by using noise management techniques. Their scheme is based on the Learning With Errors (LWE) problem. Later, the BGV scheme was implemented as a software library HElib
using C++. The library was used to re-implement the homomorphic evaluation of an earlier AES circuit [16] by Gentry, Halevi and Smart. They achieved an amortized time of 2 seconds for 120 blocks of AES implementation. Later, [2] presented a new tensor product technique that reduces the noise from quadratic to linear growth. The technique is applicable to LWE schemes, i.e. BGV style schemes. Later, López-Alt, Tromer and Vaikuntanathan (LTV) [23] proposed an FHE scheme based on a variant of NTRU [28] that has multi key support. Doröz, Hu and Sunar implemented the proposed LTV scheme and using it evaluated a custom AES circuit and a level optimized Prince block cipher circuit [10, 11] homomorphically. These implementations were later accelerated using a GPU by Dai et. al. [7, 8]. With GPU support, amortized timings of homomorphic Prince and AES evaluations reduced to 24 msec and 7.3 sec respectively. Recently, a new approximate eigenvector FHE scheme with reduction to LWE was proposed by Gentry, Shai and Waters (GSW) [17]. The approximate eigenvector, eigenvalue pairs are used in the construction and they introduce a new noise management technique called flattening. GSW is asymptotically faster due to the use of standard matrix operations in order to apply homomorphic addition and multiplications. With flattening the need for costly relinearization operations and any association storage of massive evaluation keys is eliminated.

**Applications.** The increasing number of new FHE schemes proposed along with a variety of optimizations, motivated researchers to conduct experiments on their practicality in applications. For example, Lagendijk et al. [20] give details on applicability of homomorphic encryption and multi-party computation for signal processing operations. These signal processing operations include but are not limited to linear filters, correlation evaluations, thresholding, signal transformations, inner product calculations and dimension reduction.

In [24], Lauter et al. focus on simple statistical operations that can be used in real-life cloud services for medical or financial applications such as finding the mean, the standard deviation and the logistical regression. Since these functions do not have high multiplicative depth, they are not necessarily required to be implemented using an FHE scheme, but an SHE construction is sufficient. In the same work, they also implement the SHE scheme of Brakerski and Vaikuntanathan [4] using Magma algebra program. The same reference discussed how to pack multiple message bits into a ciphertext. As noted, even though it is possible to pack multiple ciphertexts into a single ciphertext, there are some problems. First of all, they state that there is no known technique to unpack the messages in the encrypted form, so they cannot retrieve the messages within a packed ciphertext. Secondly, arithmetic operations become limited, i.e. we cannot perform multiplication without destroying the messages in the ciphertext.

Later, in [21] Lauter et al. investigate another homomorphic application: genomic data computation algorithms. They measure the performance of algorithms such as Pearson Goodness-of-Fit test, the D’ and r²-measures of linkage disequilibrium, the Estimation Maximization algorithm for haplotyping, and the Cochran-Armitage Test for Trend. Another homomorphic encryption application on medical data is performed in [1] by Bos et al. The technique is applied on
medical data to perform private predictive analysis on the probability of cardiovascular disease.

There are many other homomorphic applications from various fields that are implemented by various groups of researchers. A machine learning algorithm, i.e. Linear Means Classifier and Fisher’s Linear Discriminant Classifier on the Wisconsin Breast Cancer Data set, is implemented in [18] by Graepel et al. Another application is dynamic programming that is presented by Cheon et al. [6]. They implemented algorithms such as Hamming distance, edit distance, and the Smith-Waterman algorithm on genomic data.

2 Motivation

The recent progress in fully homomorphic encryption schemes motivated researchers to investigate applications of FHE schemes as solutions to real life privacy concerning problems. In these applications, researchers face difficulties to evaluate some of the basic primitive operations homomorphically. The lack of these homomorphic primitive operations limits the applications or forces protocol changes, e.g. by moving some of the more difficult operations to the client side. In this work, we focus on two different problems. The first one is a remarkably important, yet still open problem of homomorphic unbatching of a single ciphertext that contains batched messages. The second one is the homomorphic evaluation of the NTT operation over multiple ciphertexts. The details are as follows:

- **Homomorphic Unbatching.** This problem was explicitly posed by Lauter et al. in [24]: How can we unpack information belonging to numerous clients packed at the beginning of a homomorphic evaluation session into a single ciphertext for efficiency. The authors mention that if there was a method for homomorphic unbatching, a server might easily batch messages of different clients on a single ciphertext, process the ciphertext and later it can homomorphically unbatch the individual results to different ciphertexts to be delivered to the respective clients. Basically, this method helps to significantly improve the computational performance on servers by compressing the different ciphertext messages from users into a single ciphertext for parallel processing. In addition it gives the option to separate these results into different ciphertexts so that the result is only send to the owner of the data. Here we show a way to achieve homomorphic unbatching by using the NTT homomorphically. We focus on ways to implement the homomorphic NTT and show the difficulties of achieving FFT speed to this end.

- **Homomorphic NTT.** In this case, we implement the homomorphic NTT using a different method and we succeed to achieve FFT speed, and with this method our goal is to compute convolutions, instead of unbatching. This operation can be used for many NTT/FFT applications, such as filtering, large integer and polynomial multiplications, Chebyshev approximation and efficient matrix-vector multiplications in the FHE setting.
Our Contribution. In this work we present an array of solutions to improve the versatility of homomorphic NTT, specifically:

- We tackle the problem of computing the Number Theoretical Transform homomorphically over the domain defined by the message space. It turns out that noise growth is a significant issue and FFT speed evaluation is difficult to achieve without homomorphic modular reduction. We work out a solution and provide concrete performance figures.
- Empowered by homomorphic NTT we define homomorphic batching / un-batching which allows us to move the coefficients of encrypted message polynomials into message slots and vice versa. Using homomorphic batching one may unpack message polynomials, i.e. extract coefficients from encrypted message polynomials; and more broadly change the processing domain on-the-fly while evaluation proceeds.
- From a security perspective, homomorphic batching / unbatching allows us to prevent information leakage through partial evaluation results that accumulate in batched messages. This is of utmost concern in multi-user settings where multiple streams of information are bundled together and processed simultaneously.
- We implement homomorphic NTT using another method in which we encrypt the elements of the NTT in different ciphertexts and perform levels of NTT computations on these ciphertexts. In the end we achieve the elements of NTT result in different ciphertexts. This way we are able to achieve the FFT speed, batch independent NTT operations for parallel processing and achieve amortized time. However we are unable to compute batch/unbatch homomorphically.
- Also, we give run-time complexity analysis on both of the proposed homomorphic NTT methods.
- Finally we note that homomorphic NTT is of independent interest to numerous applications, e.g. filtering in digital signal processing, spectral decomposition and analysis, etc.

3 FHE Background

In this work, we use customized leveled FHE implementation proposed by Doröz, Hu and Sunar (DHS) [10]. The library is written in C++ and it uses NTL software with GMP support. The library supports the leveled multi-key FHE scheme implementation proposed in 2012 by López-Alt, Tromer and Vaikuntanathan (LTV) [23]. It is based on a variant of Stehlé and Steinfeld’s [27] NTRU encryption with new operation called relinearization and existing operation modulus switching to control noise. Although the scheme can support support multi-keys (users), the implemented library focuses on the single-key (user) scenario.

The LTV scheme uses the following primitives: there is a polynomial ring \( \mathbb{R}_q = \mathbb{Z}_q[x]/(x^N + 1) \) with \( N \) being the polynomial degree and \( q \) being the prime modulus. The message space is defined using a prime modulus \( p \). In the scheme,
special ring structure is used to promote the evaluation keys to the next level selected as a power of a fixed prime, i.e. two operations as follows: control the noise; relinearization and modulus switching. We summarize these in terms of noise growth. The scheme uses two significant operations to the noise level of the ciphertexts and multiplication explicitly outweighs addition of messages in the ring $\mathbb{Z}_p$. We compute the evaluation keys as to compute the secret keys to the noise size at each level. We compute the evaluation keys as $\tilde{f}^{(i)}(x) = h^{(i)}s^{(i)} + pe^{(i)} + 2^pf^{2(i-1)}$ where $\{s^{(i)}, e^{(i)}\} \leftarrow \chi$ and $\rho \in [0, \lceil \log (q_i) \rceil]$ for each level $i$.

**Encrypt.** We encrypt a message $m = e^{(i)}f^{(i)}$ (mod $p$).

**Decrypt.** The decryption for $i^{th}$ level is simply achieved by evaluating: $m = \tilde{e}^{(i)} = h^{(i)}s^{(i)} + pe^{(i)} + b$ by sampling $\{s^{(i)}, e^{(i)}\} \in \chi$ for $i^{th}$ level.

**KeyGen.** In the scheme modulus $q$ is a decreasing sequence of prime numbers for each level; $q_0 > q_1 > q_2 \cdots > q_d$. We select the prime modulus $q_i$, according to the noise size at $i^{th}$ level. We sample two polynomials $g^{(i)} \leftarrow \chi$ and $u^{(i)} \leftarrow \chi$ to compute the secret keys $f^{(i)} = pu^{(i)} + 1$ and the public key $h^{(i)} = pg^{(i)}f^{(i)}$ for each level. We compute the evaluation keys as $\tilde{f}^{(i)}(x) = h^{(i)}s^{(i)} + pe^{(i)} + 2^pf^{2(i-1)}$.

**Modulus Switching.** This operation is a way of reducing the existing noise in the ciphertexts. Basically, we perform $\tilde{e}^{(i)}(x) = \left\lfloor \frac{\tilde{e}^{(i-1)}(x)}{q_i} \right\rfloor$ on each coefficient of the ciphertext. We achieve following two things: a reduction in the noise by $\log (q_i/q_{i-1})$ bits and a new field $\mathbb{Z}_{q_i}$ for modular arithmetic. The ceil/floor operation $\lfloor \cdot \rfloor_p$ refers to rounding to match the parity for modular $p$. An advantage of the scheme is that its performance is increased as we switch levels due to a smaller modulus $q_i$.

**Relinearization.** This operation is necessary after each multiplication operation in order to prevent the noise growth and the increase of inverse powers of secret keys $f^{(i)}$. Simply we are switching square power of secret key $f^{-(i-1)}$ for level $i - 1$ with the new secret key $f^{(i-1)}$ of level $i$. We evaluate the relinearization operation by computing: $\tilde{e}^{(i)}(x) = \sum_{\rho} \tilde{f}^{(i)}_{\rho}(x)s^{(i-1)}(x)$. In the equation $\tilde{f}^{(i-1)}_{\rho}(x)$ are binary polynomials that forms $\tilde{e}^{(i-1)}(x) = \sum_{\rho} 2^\rho \tilde{e}^{(i-1)}_{\rho}(x)$.

**Specializations.** In DHS library [10], the decreasing modulus sequence is selected as a power of a fixed prime, i.e. $q_i = \sigma^{k-1}$. Here the prime $\sigma$ is equal to the noise cutting size for each level and $k$ is the circuit depth plus one. This special ring structure is used to promote the evaluation keys to the next level.
using modulo reduction when needed, i.e. $c_p^{(i)}(x) = c_p^{(0)}(x) \mod q_i$. This reduces the key size significantly, as we only need to store the first level evaluation keys.

4 NTT Background

In this section, we will briefly go over Fourier Transform (FT) and its finite version, Discrete Fourier Transform (DFT) which is more widely used in practical applications. Then, we will talk about NTT which is a variant of DFT.

4.1 Fourier Transform

Fourier Transform is a signal transformation method that is used in many mathematical and scientific applications such as filtering, time domain and frequency domain conversions, large integer multiplications and sine/cosine wave transformations. For practical applications, DFT, the finite version of the FT, is used. If we have a sequence of $N$ complex numbers $x_0, \cdots, x_{N-1}$ in one domain, applying DFT will give us a new sequence of $N$ complex numbers $X_0, \cdots, X_{N-1}$ in another domain and these values can be computed by simply evaluating:

$$X_k = \sum_{j=0}^{N-1} x_j e^{-\frac{2\pi i j k}{N}}, \quad \forall k \in [0, N - 1].$$

This linear transformation can be represented using a transformation matrix. We can define the same operation as a multiplication of our input vector $\vec{x} = [x_0, \ldots, x_{N-1}]$ with a special transformation matrix $\mathbf{W}$, i.e. $\vec{X} = \mathbf{W} \cdot \vec{x}$. We have the output vector $\vec{X} = [X_0, \ldots, X_{N-1}]$ of length $N$. The transformation matrix $\mathbf{W}$ has the structure of a Vandermonde matrix with entries $\alpha_{k,j} = (\alpha^k)^j$ where $\alpha = e^{\frac{2\pi i}{N}}$ and can be visualized as follows:

$$
\begin{bmatrix}
\alpha^0 & \alpha^0 & \alpha^0 & \cdots & \alpha^0 & \alpha^0 \\
\alpha^0 & \alpha^1 & \alpha^2 & \cdots & \alpha^N-2 & \alpha^{N-1} \\
\alpha^2 & \alpha^3 & \alpha^4 & \cdots & \alpha^{N-2} & \alpha^{N-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\alpha^{(N-2)} & \alpha^{(N-2)+1} & \alpha^{(N-2)+2} & \cdots & \alpha^{(N-2)(N-2)} & \alpha^{(N-2)(N-1)} \\
\alpha^{(N-1)+0} & \alpha^{(N-1)+1} & \alpha^{(N-1)+2} & \cdots & \alpha^{(N-1)(N-2)} & \alpha^{(N-1)(N-1)}
\end{bmatrix}
$$

In a naïve implementation the time complexity of the DFT becomes $O(N^2)$. However, by using a Fast Fourier Transform (FFT) algorithm namely Cooley-Tukey method, we can reduce the cost of the evaluation to $O(N \log (N) \log \log (N))$. The Cooley-Tukey algorithm is based on re-expressing the DFT equation into summation of two sub-DFT equations:

$$X_k = \sum_{j=0}^{N-1} x_j e^{-\frac{2\pi i j k}{N}} = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i j k}{N/2}} + e^{-\frac{\pi i k}{N/2}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i j k}{N/2}}$$

Even

Odd
As shown on the equation above, summation on the left calculates the DFT of the even indices and the summation on the right calculates the DFT of the odd indices. These odd/even DFT summations can also be re-expressed as summation of sub-odd and -even indicies. This procedure can be applied recursively until the DFT size is small enough to be evaluated fast enough. Later, the FFT can be calculated by reconstructing the calculated sub-DFT’s by going into upper levels in the recursive function.

4.2 Number Theoretic Transform

Number Theoretic Transform is a specialization of DFT over the ring $\mathbb{Z}/p\mathbb{Z}$ by replacing $e^{-i2\pi k/N}$ with a primitive $N$th root of unity $\omega$. One of the most common usage of the method is to evaluate large integer or polynomial multiplications. It prevents the errors that might be caused by the floating point arithmetic of FFT and provides precise arithmetic evaluations. We can compute the NTT by simply evaluating:

$$X_k = \sum_{j=0}^{N-1} x_j \omega^{kj} \pmod{p},$$

where $\mathbf{x}$ is again the input vector, $p$ is the prime modulus and $k \in [0, N-1]$. The inverse-NTT of the evaluated vector $\mathbf{X}$ is computed using the same equation by replacing $\omega$ with $\omega^{-1} \pmod{p}$:

$$x_k = \sum_{j=0}^{N-1} X_j \omega^{-kj} \pmod{p}.$$

Thus the transformation matrix $W$ and the inverse transformation matrix $W^{-1}$ becomes:

\[
\begin{bmatrix}
1 & 1 & \ldots & 1 \\
1 & \omega & \ldots & \omega^{N-1} \\
\vdots & \vdots & \ddots & \vdots \\
1 & \omega^{N-1} & \ldots & \omega^{(N-1)(N-1)}
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
1 & 1 & \ldots & 1 \\
1 & \omega^{-1} & \ldots & \omega^{-(N-1)} \\
\vdots & \vdots & \ddots & \vdots \\
1 & \omega^{-(N-1)} & \ldots & \omega^{-(N-1)(N-1)}
\end{bmatrix}
\]

respectively. Using the Cooley-Tukey approach that is explained in the previous section, the NTT conversion can achieve a runtime of $O(N \log(N) \log \log(N))$.

5 Homomorphic NTT

In this section we give two methods to perform homomorphic NTT and discuss their advantages and disadvantages. In the first method we show that we are able to homomorphically batch/unbatch messages on a single ciphertext, but also show that it has limitations to achieve FFT speed. In the second method we show that we can overcome the problem and achieve FFT speed, but we need use $N$ ciphertexts for input and output.
5.1 Homomorphic Batching/Unbatching

Batching is a powerful data encoding technique that is used for processing independent data in parallel. It, when used in homomorphic computing, yields great versatility in the computations and greatly improves performance. Although batching is important in homomorphic computing, existing implementations used batching only to process independent data in parallel. Therefore, these implementations perform batching only before the encryption and after the decryption of the messages as follows. First, many independent data is embedded into message slots. The message slot contents are then encoded into a polynomial representation with the help of the (inverse) Chinese Remainder Theorem (CRT). The encoded message polynomial is then encrypted. Once batched messages are encrypted, then they are processed in independent homomorphic evaluation paths, i.e., evaluation of many AES encryption (by batching) using single polynomial ciphertext. Once the evaluations are completed, the output message polynomial is decrypted and the message slot contents are retrieved using the CRT residue computation.

Here, we want to extend the capabilities of homomorphic encryption by implementing a batching technique homomorphically so that we are able to batch any data on-the-fly. In order to do that, we need a transformation that is capable to bring the message slot contents into the coefficients in the polynomial representation and back while all data is maintained in encrypted form. This homomorphic arithmetic actually is the equivalent of evaluating a CRT and its inverse homomorphically. We define this as homomorphic batching as follows.

**Definition 1 (Homomorphic Batching).** The isomorphism $\mathbb{Z}_p[x]/(\Phi(x)) \cong \mathbb{Z}_p[x]/(x - \zeta) \times \mathbb{Z}_p[x]/(x - \zeta^2) \times \cdots \times \mathbb{Z}_p[x]/(x - \zeta^{N-1})$ where $\Phi(x) = \prod(x - \zeta^i)$ denotes the characteristic polynomial in $\mathbb{Z}_p$ of degree $N - 1$ and $\zeta$ denotes a primitive $N$th root of unity in $\mathbb{Z}_p$. We refer to the homomorphic evaluation of the isomorphism and its inverse as homomorphic unbatching and batching, respectively.

From a computational perspective, the encoding/decoding operations both amount to the evaluation of a linear transformation on the message slot/polynomial coefficients, respectively. For instance, the message polynomial $m(x)$ is decoded as $\bar{m} = [m(\zeta), m(\zeta^2), \ldots, m(\zeta^{N-1})]$. The encoding function may be computed, for example, using Lagrange interpolation. Computations may be expressed as linear transformations as:

$$\text{Decode}(m(x)) = \bar{m} = W\bar{m} \quad \text{and} \quad \text{Encode}(\bar{m}) = m = W^{-1}\bar{m},$$

which $[W]_{ij} = \zeta^{ij} \in \mathbb{Z}_p$ and $\bar{m}$ is a vector that holds the coefficients of $m(x)$. The operation appears simple enough since modulo $p$ operations is the natural domain of the homomorphic evaluations and since all we need to compute is constant multiplications by powers of $\zeta$. Typically, when batching in cleartext we compute the encoding and decoding operation with the aid of an $N$th root of unity $\zeta \in \mathbb{Z}_p$ via a number theoretical transform (NTT) to gain FFT speed, i.e., $O(N \log(N))$ encoding/decoding performance. However, using cyclotomic
polynomials to batch messages bring limitations for us to directly evaluate NTT and achieve FFT speed.

**Limitations.** The batched messages in a polynomial presents independent computation paths. However, when we compute it’s linear transformation we need to sum the scaled message slot contents. Thus we need a means to move the message slot contents. We achieve this by using \( \Phi(x) = \prod_{i \in [N-1]} (x - \zeta^i) \) where \( b \) is a primitive element of \( \mathbb{Z}_q \) and later by evaluating \( m(x^b) \), we rotate the message slot contents. The side-effect of this shift operation on a ciphertext is that the key is altered during the evaluation process:

\[
c(x^b) = pg(x^b)s(x^b)f^{-1}(x^b) + pe(x^b) + m(x^b)
\]

The ciphertext will still decrypt correctly since \( g(x^b), s(x^b) \) and \( c(x^b) \) will have small norm. However, to decrypt the ciphertext the key needs to be updated to \( f(x^b) \). To restore the original key we may use key switching \( \text{L.KeySwitch}(c(x^b), \theta) \) where \( \theta = \{ \text{L.Encrypt}(w^\tau f(x^b)_i) \} \) for \( \tau \in [\log q] \). With this approach we can rotate the message slot contents an arbitrary \( i \) positions by evaluating the ciphertext polynomial as \( c(x^{\tau i}) \) and then by applying a key switching operation with \( f(x^{\tau i}) \).

Here the problem lies with the selection of cyclotomic polynomial \( \Phi(x) \) as the modulus. It gives a decoding matrix \( W \) as:

\[
\begin{bmatrix}
\alpha^0 & \alpha^1 & \alpha^2 & \ldots & \alpha^{N-2} & \alpha^{N-1} \\
\alpha^{2\cdot0} & \alpha^{2\cdot1} & \alpha^{2\cdot2} & \ldots & \alpha^{2\cdot(N-2)} & \alpha^{2\cdot(N-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\alpha^{(N-2)\cdot0} & \alpha^{(N-2)\cdot1} & \alpha^{(N-2)\cdot2} & \ldots & \alpha^{(N-2)\cdot(N-2)} & \alpha^{(N-2)\cdot(N-1)} \\
\alpha^{(N-1)\cdot0} & \alpha^{(N-1)\cdot1} & \alpha^{(N-1)\cdot2} & \ldots & \alpha^{(N-1)\cdot(N-2)} & \alpha^{(N-1)\cdot(N-1)}
\end{bmatrix}
\]

and an encoding matrix \( W^{-1} \) (mod \( p \)). The formed matrices \( W \) and \( W^{-1} \) of \( \Phi(x) \) are not Vandermonde matrices, therefore we are unable to apply Cooley-Tukey’s algorithm. Since we cannot apply the even-odd splitting trick, we are unable to apply fast NTT.

We can solve \( W \) and \( W^{-1} \) not being Vandermonde matrices by switching the cyclotomic polynomial \( \Phi(x) \) with \( x^N - 1 \) which has the following form:

\[
x^N - 1 = (x - 1) \cdot \Phi(x) = \prod_{i=0}^{i=N-1} (x - \zeta^i),
\]

where \( N \) is power of 2. This converts the batching operation to be applicable using Vandermonde matrix multiplication which is suitable for fast NTT using Cooley-Tukey. Although the scheme is suitable for fast NTT, we are not able to rotate the messages as in cyclotomic polynomials. The message in the first slot, i.e. in \( (x - \zeta^0) \), never rotates in function \( f(x^N) \) for any \( i \).
Homomorphic Batch/Unbatch. With the issues addressed above, we are able to compute homomorphic unbatching by the following equation:

$$L_{\text{Unbatch}}(c) = \sum_{s \in \mathbb{N}} c^{(s)} \cdot \text{Encode}((W^{\text{rot}})^T[s]).$$

Here $c^{(s)}$ represents the rotated versions of the ciphertext (and message) coefficients by $s$ positions. $W^{\text{rot}}$ is the transformation matrix where each row index $i$ is rotated by $i$. The symbol $\top$ represents the transpose of the matrix and $[s]$ is used for the row index $s$ of the matrix. In case of batching we only replace $W$ with $W^{-3}$. The all operation requires only one level of circuit depth for evaluation.

Packing/Unpacking. Packing and unpacking messages in homomorphic encryption is useful for processing the information in parallel by batching/unbatching multi-user information. In multi-user scenarios where we have many users that provides input for a process, we can pack the informations to efficiently process. In word message space we can input $N$ user information into the same ciphertext that will provide $N$ times speedup for information processing. In [24] Lauter et al. show how to pack the messages from multiple ciphertexts into one ciphertext. They mention that they cannot present a technique to unpack messages which restricts their computations. After a packing operation the polynomial multiplications rounds and deforms the information because of the polynomial modulus. Our main motivation here was to transfer the message slot contents into the polynomial coefficients and back. However, we achieve unpacking, which is regarded as difficult to achieve, with the aid of homomorphic batching. We may unpack the $k^{th}$ coefficient $c^{(k)} = L_{\text{Encrypt}}(m_k x^k)$ and $c = \left(\sum_{i \in [N]} m_i x^i\right)$ with the following steps:

- Push coefficients into the message slots $\tilde{c} = L_{\text{Unbatch}}(c) \in \mathcal{R}$,
- Filter desired coefficient(s) by multiplying with constant cleartext mask $\mu_k(x) = \text{NTT}^{-1}(I_k)$,
- Push message slot contents back into coefficients by homomorphic batching $c^{(k)} = L_{\text{Batch}}(\mu_k(x)) \in \mathcal{R}$.

The packing/unpacking operation enables to do privatization in the homomorphic encryption. We may easily batch the information for parallel processing and later send the result informations for filtering the results for each users in multi-user scenarios. This prevents the information leaks while returning the results to the users, since we are able to eliminate the results of other users from the ciphertext.

5.2 Homomorphic NTT Using Parallel Batching

There is an alternative and straightforward way to implement homomorphic NTT that is not limited by the issues given in the previous section. We can
encrypt each message to be used in the NTT separately: $c_i = hs_i + pe_i + m_i$. Then, we can compute the fast NTT using the Cooley-Tukey algorithm as:

$$C_k = \sum_{j=0}^{N-1} c_j \zeta^{jk} = \sum_{j=0}^{N/2-1} c_{2j} \zeta^{2jk} + \zeta^k \sum_{j=0}^{N/2-1} c_{2j+1} \zeta^{2jk}$$

where $\zeta$ is a primitive $N^{th}$ root of unity modulo $p$. Since each message is in an independent ciphertext, we can easily divide them into even and odd indices. This way we can easily compute the fast NTT of the input. However there are two main issues with the scheme that limits the operation:

- The modulo $p$ reduction does not take place until the very end of the decryption step, i.e. $L_{\text{Decrypt}}(c) = [cf]_q \pmod{p}$. Therefore, intermediate results will accumulate powers of $\zeta$, which likely will cause a wraparound and decryption failure. One alternative is to aggressively apply noise reduction, e.g. modulus switching, even for the constant multiplications. However, this will increase the evaluation levels significantly. For instance, even a moderate $N = 2^{13}$ would add 13 evaluation levels. To overcome noise accumulation we abandon FFT style evaluation and instead only multiply with precomputed $W, W^{-1} \in \mathbb{Z}_p^{(N-1) \times (N-1)}$.

- The number of ciphertexts increase to the number of NTT elements, i.e. $N$ in our case, from a single ciphertext. This increases the ciphertext input size by $N$ times and it is equal to $N^2 \log q$. More than the computational complexity, it increases the I/O transactions of the scheme significantly. Although we have $N$ ciphertexts at the end, we can simply batch them by evaluating:

$$\sum_{i=0}^{N-1} C_k \cdot x^i.$$

This solves the issue of having many ciphertexts. However, we are unable to unbatch the values in the equation which limits further processing. We can access the values individually only after a decryption operation.

Although we are not able to batch the dependent elements in a fast NTT operation, we are able to batch $N$ independent fast NTT operations. Basically, we are able to use the empty message slots to evaluate $N$ parallel fast NTT operations. This way we are able to achieve an amortized time that is $N$ times better than the total runtime.

6 Complexity Analysis

Here we discuss the complexity of the two proposed algorithms. In homomorphic batching we need to compute $N$ multiplications of ciphertext with a polynomial formed by the row values of $W$. Using a large polynomial multiplication algorithm, such as Schönhage-Strassen algorithm, we achieve a run-time complexity
of $O(N \log N \log \log N)$. Furthermore, we have to perform key-switching operations to the ciphertexts to correct the public keys that are corrupted in rotation operation. This is a similar operation to the relinearization, so we can apply the time complexity of relinearization in [10] for key-switching as well. We have $N$ key-switching operations with run-time complexity of $O((\log q) N \log N \log \log N)$. In total the algorithm has a run-time complexity of $O((\log q) N^2 \log N \log \log N)$.

In the second algorithm, i.e. homomorphic NTT using parallel batching, we have $\log N$ stages of NTT operations. Each stage $N$ multiplications of a constant with a ciphertext which makes $N^2$ coefficient multiplications per stage. In total the algorithm has run-time of $O(N^2 \log N)$.

An important thing to note is that the complexity analysis takes into account only the number of coefficient multiplications. It does not include the run-time complexity of the coefficient multiplications. In the first case we have small and fix size coefficients which gives an advantage in real time applications against the second method. The second method has larger coefficients because of the leveled implementation. Thus it takes longer time to process the second method even though the run-time complexity of the method is smaller in terms of number of coefficient multiplications.

7 Implementation Results

We implemented the algorithms using a leveled LTV scheme using Shoup’s NTL library version 9.0 [25] compiled with the GMP 5.1.3 package. For parameter selection we utilized the two Hermite factor analysis using the formula in [22], i.e. $1.8/\log \delta - 110$. The security level of the experiments varies on the settings, but each setting has at least 100-bit security.

In the homomorphic NTT using homomorphic batching we use special cyclotomic polynomial $\Phi_m(x)$, where we set $m$ as a prime number to have $\Phi_m(x) = x^m + x^{m/2} + \cdots + x^2 + x$, to perform faster modular reduction. The results are summarized in Table 1. In the algorithm we have one constant polynomial multiplication and $N$ additions, so our prime modulus $q$ does not grow too large. The values of $N$ are chosen to be close to as powers of two, i.e. 2048, 4096, 8192. The message slots are used for the same NTT operation so there is no amortized time.

In the second case, we compute homomorphic NTT by using parallel batching. We choose the polynomial degree $N = 16384$ and modulus bitsize $\log q = 512$ which are slightly higher values compared to the first algorithm. The reason behind this is that we need to handle the noise in stages, so the modulus $q$ grows significantly. Our implementation achieves a runtime of 108 minutes. Since we are able to batch $N$ independent homomorphic NTT computation, we achieve 0.4 second of amortized time.
<table>
<thead>
<tr>
<th>( N )</th>
<th>( \log q )</th>
<th>Security (in bits)</th>
<th>Total Time (in minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2080</td>
<td>64</td>
<td>140</td>
<td>2.5</td>
</tr>
<tr>
<td>4252</td>
<td>64</td>
<td>400</td>
<td>10.7</td>
</tr>
<tr>
<td>8782</td>
<td>64</td>
<td>943</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 1. Timings for Homomorphic Batching/Unbatching operation.

8 Conclusion

To improve the versatility of homomorphic encryption applications, we tackled another challenging problem, i.e. the problem of moving data in encrypted form from the message slots into the message polynomial coefficients and back. We called this operation homomorphic batching/unbatching. Via homomorphic batching one can extract coefficients and achieve unpacking operations easily. In addition, the batching operation enabled via a homomorphic NTT operation, which will be of interest for numerous signal processing applications.

References

Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation

Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin Butler, and Patrick Traynor

1 Yale University. E-mail: {debayan.gupta, joan.feigenbaum}@yale.edu
2 University of Florida. E-mail: bmood@ufl.edu, {butler, traynor}@cise.ufl.edu

Abstract. Recent developments have made two-party secure function evaluation (2P-SFE) vastly more efficient. However, because they make extensive use of cryptographic operations, these protocols remain too slow for practical use by most applications. The introduction of Intel’s Software Guard Extensions (SGX), which provide an environment for the isolated execution of code and handling of data, offers an opportunity to overcome such performance concerns. In this paper, we explore the challenges of using SGX to achieve security guarantees similar to those found in traditional 2P-SFE systems. After demonstrating a number of critical concerns, we develop two protocols for secure computation in the semi-honest model on this platform: one in which both parties are SGX-enabled and a second in which only one party has direct access to this hardware. We then show how these protocols can be made secure in the malicious model. We conclude that implementing 2P-SFE on SGX-enabled devices can render it practical for a wide range of applications.

1 Introduction

Secure Function Evaluation (SFE) is a powerful way to protect sensitive data. Made possible by a range of cryptographic primitives, SFE allows multiple parties to compute the output of a function without revealing the potentially sensitive inputs of any individual party. In this paper, we focus on the case of two-party secure function evaluation (2P-SFE). While both the performance of and the security provided by these underlying primitives have improved dramatically over the past decade [43, 8, 16, 24, 26, 28, 32], the expense of using 2P-SFE remains too high for most practical applications.

An emerging hardware primitive may help to reduce the cost of such computation substantially. Intel’s Software Guard Extensions (SGX) [1, 25] provide a module within upcoming chipsets that allow for the creation of secure containers called “enclaves.” These hardware-enforced sandboxes allow for code and data to be executed without the influence of code running in the traditional registers of the processor. In addition, an SGX system can use hardware-based attestation to prove that an enclave performs the operations as claimed. While not necessarily appropriate for all scenarios, this set of capabilities may help to support the use of fast and strong 2P-SFE in a wide range of practical applications.
In this paper, we perform the first analysis of SGX as a platform on which to implement 2P-SFE. Beginning with a tutorial example, we show why the naive execution of functions within SGX fails to provide the strong properties necessary to prevent significant leakage. From this observation, we then make the following contributions:

– We show how to augment an SGX system to provide stronger guarantees against leakage and provide a protocol that enables two SGX systems to perform 2P-SFE more efficiently than a pure garbled-circuits implementation. We refer to this approach as SGX-supported 2P-SFE. We then provide a protocol for securely outsourcing the SGX-supported 2P-SFE computation from a resource constrained device (i.e., one without an SGX module) to an SGX-compliant device (i.e., another device that has an SGX module). This allows us to take advantage of a remote SGX hardware unit without requiring universal deployment.

– We show how to modify 2P-SFE protocols secure against semi-honest adversaries so that, when run on augmented SGX machines, they are secure against malicious adversaries.

– We describe a number of novel use cases for SGX with our augmentations.

The rest of the paper is organized as follows: Section 2 provides background on 2P-SFE and SGX. Section 3 explains problems that arise in straightforward attempts to use SGX for 2P-SFE. Section 4 describes how to augment SGX so that it can be used to implement 2P-SFE, a secure-outsourcing protocol for non-SGX machines, and how 2P-SFE and SGX can be used efficiently in conjunction to provide better security. Section 5 discusses previous work on secure-execution environments, and Section 6 provides conclusions and open questions.

2 Technical Background

We begin with a brief overview of garbled-circuit 2P-SFE and SGX. We use this as a point of departure for our investigation of SGX-based protocols for 2P-SFE and why they are harder to design than one might imagine at first glance.

2.1 Garbled Circuits for Two-Party, Secure Function Evaluation

In a garbled-circuit protocol, two parties with private inputs jointly compute a function represented as a Boolean circuit. Both parties receive outputs – the scenario described in Section 1, which has a single output $y$ for both parties, is a special case; in general, the protocol may deliver different outputs to each party. First, a compiler [32,37] is used to convert the function into a Boolean circuit. One of the parties, the generator, encrypts, or garbles the Boolean circuit. He then sends it to the evaluator, who evaluates the garbled circuit without learning any information about the generator’s inputs, intermediate values (i.e., those computed by non-output gates of the circuit), or the generator’s output. Finally, the evaluator sends the generator’s (encrypted) output back to him.
Each gate in a Boolean circuit can be evaluated using its truth table to get the output corresponding to the input values. Likewise, a garbled circuit is made up of many garbled gates, and each gate is evaluated in turn. A garbled gate’s output entry in the truth table is encrypted under a unique combination of the two inputs: \( TT_{i,j} = Enc(X_i, Y_j) \oplus Out_{i,j} \), where \( TT_{i,j} \) is the truth-table entry created by the \( i^{th} \) value of wire \( X \) and the \( j^{th} \) value of wire \( Y \), and \( Out_{i,j} \) is the corresponding unencrypted output value. The truth-table entries are permuted so that the position of the (only) decryptable entry does not leak the underlying Boolean value. Once the evaluator receives the garbled gates and the input values, she finds the correct garbled output by trying to decrypt each truth-table entry or by using the point-and-permute optimization \[^{32}\].

There are two basic types of adversaries in the garbled-circuit literature: semi-honest and malicious adversaries; each captures a basic threat model. (There exist others, such as the covert model, but we do not discuss them here.) Semi-honest adversaries faithfully follow the protocol but attempt to gain information by observing all transmitted messages. Malicious adversaries, on the other hand, may behave in any arbitrarily manner in an attempt to gain information about another party’s input or output, to corrupt the computation (i.e., to cause incorrect outputs), or to block the protocol execution from completing.

To achieve security against malicious adversaries, the computation must be performed \( N \) times in order to prevent the generator from creating an incorrect circuit. The security parameter \( N \) sets the upper bound on an adversary’s successfully cheating at \( \frac{1}{2^N} \). There must be mechanisms to ensure that the same inputs are used each time and a way to ensure the evaluator does not corrupt the generator’s output. These are solved problems in the garbled-circuit literature.

2P-SFE and garbled circuits were introduced in the seminal paper of Yao \[^{50}\], and the area has since been studied extensively by the cryptography community. One very notable achievement was the creation of the first general-purpose 2P-SFE platform, Fairplay \[^{32}\]. Today, many 2P-SFE platforms exist \([43, 8, 16, 24, 26, 28, 31, 36]\), and their performance is improving. Such platforms have been used for scenarios as varied as those of farmers conducting beet-root auctions \[^{7}\], inter-domain routing \[^{23}\], governments reporting aggregated salary data \[^{6}\], and database policy compliance \[^{14}\]. For a detailed explanation of many essential garbled-circuit techniques, see Kreuter et al. \[^{28}\] and Perry et al. \[^{40}\].

### 2.2 Intel’s Software Guard Extensions Module

The Software Guard extensions (SGX) module allows parts of programs to be executed inside of separate segments of the CPU called enclaves. This is a general-purpose module (unlike, say, a DRM module). SGX provides a hardware-based guarantee that the programs and memory inside an enclave cannot be read or modified from outside of the enclave (including by a program in a different enclave). In particular, neither root nor any other type of special-access program can read or modify the memory inside an enclave. Technically, the data inside of an enclave are still within the same registers and cache as other programs; however, SGX processors provide functionality to prevent unauthorized access.
An adversary should not be able to determine what is accessed inside of the enclave or what is written back to RAM when the cache is full. Therefore, any data in the enclave that must be written back to main memory is encrypted and signed so that it cannot be read or modified by another program. Modifications of code, data, or stack outside an enclave cannot interfere with the operation of the enclave except in one way: If something needed by a program in the enclave is simply unavailable or has been corrupted, then the program may have to abort.

Comprehensive overviews of SGX can be found in Intel’s whitepapers \cite{1,25}. Design of systems and protocols that make extensive use of SGX is covered by, e.g., Baumann \textit{et al.} \cite{5} and Schuster \textit{et al.} \cite{42}.

### 2.3 Towards Using Secure Hardware for Garbled-Circuit Protocols

Both garbled circuits and SGX are designed for scenarios in which parties have private input data for a computation in which they want to receive the result of the computation while no one else learns either the input or the result. Therefore, it is natural to consider using SGX-enabled machines to execute a garbled-circuit protocol. The reason that it is not straightforward to do so is that garbled circuits and SGX use different techniques to protect private inputs.

In garbled-circuit protocols (and SFE more generally), cryptographic guarantees are used to ensure the privacy of the data. In SGX, users rely on secure hardware to guarantee data privacy. SGX provides security against malicious adversaries as long as one trusts Intel’s setup process. In the SFE world, this is comparable to having a trusted setup, on top of which one runs one’s protocol (here, part of the “setup” occurs at the Intel factory when the hardware and private key are created). The security properties of the exact model used by SGX are described in Intel’s whitepapers \cite{1,25}.

### 3 Why Simple “Solutions” Do Not Quite Work

The security guarantees provided by SGX do not immediately translate into being able to perform 2P-SFE protocols in general or even garbled-circuit protocols in particular. Simple solutions that use unmodified SGX primitives may leak information or, in some cases, undermine the security of other code running under SGX. In this section, we explain how that can happen.

#### 3.1 A simple 2P-SFE protocol implemented with SGX

Below, we describe a naive, straw-man protocol for performing SGX-supported 2P-SFE. There exist numerous ways of doing this, but almost all of them suffer from a number of problems that we discuss in the next subsection.

**Setup:** We start with the standard 2P-SFE setup – two mutually distrustful parties with private inputs who wish to jointly compute a function and produce private results. In this scenario, both parties have SGX-enabled machines and
have agreed to run a specific program. The two parties are as follows: the evaluator, who will use his SGX module to evaluate the program, and the sender, who will check the agreed-upon program and then send her input. In the following, a superscripted “+” denotes a public key, while a superscripted “−” denotes a private key that does not leave the SGX enclave.

Protocol

1. The sender ensures the evaluator will evaluate the correct program, $prog_{\text{sgx}}$, by checking the signed measurement, $Ecv_{\text{eval}}^{\text{measure}}$, from the evaluator’s enclave. $Ecv_{\text{eval}}^{\text{measure}}$ is signed by the evaluator enclave’s private key $Ecv_{\text{eval}}^{\text{key}}$.
2. The sender encrypts her input, $input_{\text{sender}}$, under the evaluator enclave’s public key, $Ecv_{\text{eval}}^{+}$, and sends it to the evaluator.
3. The sender’s encrypted input, $Enc(input_{\text{sender}})$ is decrypted inside of the evaluator’s enclave using $Ecv_{\text{eval}}^{-}$.
4. The evaluator enters his own input, $input_{\text{eval}}$, into the enclave.
5. The enclave puts $input_{\text{sender}}$ and $input_{\text{eval}}$ into the SGX program, $prog_{\text{sgx}}$. It then executes $prog_{\text{sgx}}$ and encrypts the sender’s output, $output_{\text{sender}}$, under the sender enclave’s public key, $Ecv_{\text{send}}^{+}$.
6. The evaluator’s enclave releases the evaluator’s output to him and sends the sender’s encrypted output, $Enc(output_{\text{sender}})$, to the sender.
7. The sender decrypts $Enc(output_{\text{sender}})$ using $Ecv_{\text{send}}^{-}$.

3.2 Problems with simple SGX-supported 2P-SFE

Side channels

1. Runtime: 2P-SFE protocols are not directly vulnerable to timing attacks. This is achieved by ensuring all program paths take equal time, at the cost of efficiency. In SGX-supported 2P-SFE, if a secret value $x$ determines the number of times, for instance, a loop is executed, the timing could easily narrow the range of $x$. Principally, an attacker could execute the same program offline with many different iterations of the same loop inside of the enclave to see how long several different numbers of iterations take. This may provide a lot of information if each iteration of the loop is easily identifiable, e.g., if each iteration takes a second to execute.
2. RAM Access: Data access is not hidden in SGX-supported 2P-SFE, which can potentially leak significant amounts of data. For example, consider a simple database-style query using a binary search, where one side, the client, sends a private query to check whether a given value exists within the database. The enclave on the server reads in the plaintext records and matches them, one by one, to the queried value. In such a scenario, the data access alone is enough to leak information about the queried value. (If the query matches, we have the value itself, and, if not, we know that the value lies within a certain range.) There exist some methods to add hardware-level cryptographic support to FPGAs [45], but not for RAM. The best ways to make RAM secure are still Oblivious RAM and similar techniques [19].
3. **RAM Timing:** A timing attack could reveal a lot of information about the item being queried in the binary search. If the item is located on the first jump, we know that it’s the value in the middle, etc.

**Cryptography vs Memory Out of Bounds** Garbled circuits rely on cryptography for data privacy; information leakage is not an issue, because we have proofs of correctness and security. While it is theoretically possible to “leak” data by simply outputting it in the predefined program, such a blatant problem is easy to notice. SGX, if used improperly, might leak information if memory goes out of bounds; this is one of the most common bugs in everyday programming [42] and can have catastrophic consequences [48, 13]. Unfortunately, in SGX, such an error would not only break the security of the program (and enclave) in question but would also affect the security of SGX as a whole, because users might be able to access or modify data that they should not be able to see.

**Trusting SGX vs Trusting Cryptography** SGX requires the users to trust that the evaluator of the program has not broken into the enclave to watch the memory and that the supply chain was not disrupted with insecure parts. These might not be acceptable assumptions for nation states or large companies. In contrast, 2P-SFE protocols provide cryptographic guarantees. They prove themselves equivalent to the “ideal model,” which uses a trusted third party. SGX uses the trusted platform model, which is weaker than the trusted third party model and allows side-channel and information flow attacks.

SGX requires us to have trust in hardware and standard cryptographic primitives (which are used by SGX to protect data), while a 2P-SFE protocol needs only the latter. Moving the “trust” from software to hardware presents additional problems – the authors are unaware of any techniques that could be used to sign and verify hardware. Given recent reports of nation states’ actively infiltrating hardware vendors at massive scales for bulk data collection, this is a major problem. Ultimately, the trust in SGX boils down to trust in hardware suppliers and whether or not the hardware can be opened and the CPU read.

4 **Using SGX for 2P-SFE Computations**

Having outlined the capabilities and limitations of SGX-supported 2P-SFE, we now present our solutions to the problems faced when trying to use SGX for 2P-SFE protocols. Throughout this section, because of space limitations, we present only short, intuitive sketches of correctness and security proofs; complete proofs will appear in a future, expanded version of the paper.

4.1 **Using SGX for 2P-SFE: Problems and solutions**

Our solution is to augment the SGX programs to prevent (or reduce) data leakage in SGX for 2P-SFE computations. These augmentations are described below.

**Timing Side Channel:** We must ensure that all code paths take approximately the same amount of time. There are many such obfuscation-based palliative mechanisms, as well as general mitigation strategies [33]. However, these
problems are more complex in some scenarios – e.g., when a secret variable determines how many times a loop executes. In this case, the time the program takes can reveal information about the value of the secret variable. It is possible to prevent any secret values from being revealed by having a fixed loop bound, but this may not always be preferable. We can limit the amount of information leaked when executing a loop by including \( N \) extra loop iterations, where \( N \) is a pseudo-random number based on secret information from both parties. Using this technique, neither party learns the number of iterations executed.

**Memory Side Channel:** We must ensure that all memory that can be touched by the SGX program is touched exactly once at the beginning of the program. Once the SGX program touches a piece of plaintext memory, the memory should not be read again unless the read is not dependent on secret information. If the read is dependent on secret information, the evaluator may be able to learn something about the secret \([17,38]\). However, if we need too much data and some are encrypted and stored outside of the enclave, there might be a correlation between when a block of memory is read and when a block of encrypted memory is sent back to RAM; for example, if a binary-search program that runs inside an enclave reads one element at a time, mere observation yields the secret query (within a range, if it is missing). In order to prevent this problem, we must ensure that a mix operation is performed to remove any correlation between plaintext memory and encrypted memory; e.g., this would occur if the memory were placed outside of the enclave in the same order as it was entered. Such mix operations, which continuously shuffle and re-encrypt data as they are accessed, already exist and are widely used to implement Oblivious RAM \([19,46]\).

**Array Out of Bounds:** To mitigate the risk of arrays out of bounds in SGX, we apply safe memory-access techniques to ensure that memory does not go out of bounds. SGX programs can use bound-checking data structures or memory-safe languages \([42]\). Although such techniques slow down the execution time of the application, both of the aforementioned methods would still be significantly faster than executing the programs in a 2P-SFE protocol.

**Cost of a 2P-SFE protocol vs SGX:** In Table 1, we note the expected cost of normal 2P-SFE using garbled circuits and SGX-supported 2P-SFE. We examine the costs of setup, input, the operation itself, data access, and memory access. As shown in the table, the primary reason for the expected improvement in the speed of SGX-supported 2P-SFE over a garbled-circuit protocol is the amount of cryptography required for each operation and data access in 2P-SFE (which is free in SGX). However, unlike garbled-circuit protocols, SGX encounters a cost to push memory out of the cache to RAM (Non-Cache Access).

### 4.2 Half and Half

With the techniques above, 2P-SFE protocols and SGX can be used together in scenarios in which parties trust each other enough to want to cooperate in the first place but not enough to release private data or blindly trust the other parties not to cheat \([29]\). However, when different groups of parties want to perform a secure computation together, a user may trust one group over another;
Table 1: Cost (in terms of cryptography) for operations in 2P-SFE and SGX-supported 2P-SFE. “-” means there is no cryptography required. \(N\) is length of input. \(C\) is length of the circuit/program. \(K\) is the bit-security parameter. \(S\) is the stat parameter (number of circuits in 2P-SFE). ¹ - per gate for 2P-SFE and per processor instruction for SGX-supported 2P-SFE. ² - the cost of saving and loading a value to or from main memory for SGX. ³ - assumes we attained a symmetric key during the setup phase and used it to encrypt the input.

<table>
<thead>
<tr>
<th></th>
<th>2P-SFE(^{Sym})</th>
<th>2P-SFE(^{Malicious})</th>
<th>SGX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sym</td>
<td>Asym</td>
<td>Sym</td>
</tr>
<tr>
<td>Setup</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Input</td>
<td>(O(N))</td>
<td>(O(K))</td>
<td>(O(N \times S))</td>
</tr>
<tr>
<td>Per Operation(^{1})</td>
<td>(O(1))</td>
<td>-</td>
<td>(O(S))</td>
</tr>
<tr>
<td>Data (array) Access</td>
<td>(O(N))</td>
<td>-</td>
<td>(O(N \times S))</td>
</tr>
<tr>
<td>Non-Cache Access(^{2})</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

We start with two companies, A and B (as shown in Figure 1), that want to perform a secure computation involving nodes both inside and outside their private networks. Parts of the computation are done inside of each company, while others require A and B to cooperate. Thus, companies could use the trust model of SGX when within their own networks and 2P-SFE when they want cryptographic guarantees instead of assuming that the hardware remains secure.

To perform such hierarchical or “mixed” SGX computations, users need to know how to convert a value from a 2P-SFE protocol to an SGX-supported 2P-SFE value and vice-versa. Once we know how to perform these transformations, we can run “mixtures” of 2P-SFE protocols and SGX. For simplicity, we deal with the semi-honest setting, although we note there are ways to do the same conversions in the malicious setting. For the purposes of this short protocol, the evaluator is the evaluator in both 2P-SFE and SGX-supported 2P-SFE. The generator is the generator for 2P-SFE and the sender in SGX-supported 2P-SFE.

Before we briefly describe the conversion process, we describe garbled circuits in more detail. During the evaluation of the garbled circuit, each wire holds an encrypted value. The generator knows the possible encrypted values (that is, which values represent 0 and 1) but does not know which value is actually on the wire (the value the evaluator has). The evaluator knows the encrypted value on each wire value but does not know what any value represents.

**Conversion from Garbled Circuit to SGX:**

1. For each garbled wire \(w_i\) we will convert to an SGX value, the evaluator has \(w_i\) (the encrypted result), and the generator has \(w_i^0\) and \(w_i^1\) (the encrypted values that represent 0 and 1).
2. The generator enters \(w_i^0\) and \(w_i^1\) into \(\text{prog}_{sgx}\) (the SGX program) as input.
Fig. 1: Half and Half. In this usage, we convert SGX-supported 2P-SFE values to standard 2P-SFE values and back in order to take advantage of the speed of the combined form when the trust model is acceptable and still allow for a stronger model when the trust model of SGX-supported 2P-SFE is not acceptable (say, the user does not trust Intel when using a public network).

3. The evaluator enters in $w_r^i$ into prog$_{sgx}$ as her input.
4. prog$_{sgx}$ calculates whether $w_r^i$ is $w_0^i$ or $w_1^i$ and sets the corresponding input, $b_i$, to match $w_r^i$.
5. prog$_{sgx}$ uses each $b_i$ as input.

Conversion from SGX to Garbled Circuit:

1. For each bit $b_i$ that will be converted into a garbled value $w_i$, the generator creates both possible garbled values, $w_0^i$ and $w_1^i$, that will represent the two possible values of $b_i$ and enters them into prog$_{sgx}$.
2. prog$_{sgx}$, based on whether $b_i$ is a 0 or 1, selects either $w_0^i$ or $w_1^i$ to be $w_r^i$.
3. Each $w_r^i$ is sent to the evaluator to be used as input to the garbled circuit.
4. The generator uses his values, $w_0^i$ and $w_1^i$, in the creation of the garbled circuit to ensure $w_r^i$ will map to a value.

Security: In order for either the generator or the evaluator to learn additional information, it has to (1) possess either $w_0^i$ or $w_1^i$ and possess $w_r^i$, or (2) see $b_i$ outside of the enclave. Since $b_i$ only exists inside of the enclave, it will not be seen by either the generator or evaluator. The generator only ever sees $w_0^i$ and $w_1^i$ and never sees $w_r^i$. Likewise, the evaluator only sees $w_r^i$ and never sees $w_0^i$ or $w_1^i$. Thus, neither party will learn any additional information.

4.3 Outsourcing

For devices that do not have an SGX module (or are slow), it would be useful to have the ability to securely outsource computation to a more powerful or better equipped system. There have already been a number of works addressing this situation in 2P-SFE [9,10,11,12,35]. In this section, we examine how we can outsource from a constrained device (that does not possess an SGX module) when we want to perform SGX-supported 2P-SFE.

In our setup, seen in Figure 2, the sender does not have an SGX unit and is outsourcing to a server, the cloud, that has an SGX unit. Any outsourcing protocol must guarantee that (1) the party we are outsourcing to (the cloud) cannot cheat, and (2) the party that performs the SGX execution (the other
Fig. 2: Outsourcing. Shows the different parties in our outsourcing protocol.

party in the original SGX-supported 2P-SFE computation, the evaluator) cannot cheat.

We assume that we are trying to protect the input and output of the sender; we also assume that the cloud and evaluator do not collude, i.e., they are not working together to corrupt the sender’s output or input. As before, superscripted “+” and “−” signs denote public and private keys, respectively.

Protocol:

1. The cloud and evaluator perform the standard SGX setup to initialize their SGX units and confirm that they are running the desired program.
2. Both parties pass enclave public keys, $E_{\text{cloud}}^{+}$key and $E_{\text{eval}}^{+}$key to the sender and authenticate by using MRSIGNER [1,25].
3. Both the evaluator and cloud enclaves send to the sender their enclave measurements, $E_{\text{cloud}}^{\text{measure}}$ and $E_{\text{eval}}^{\text{measure}}$.
4. The sender checks that $E_{\text{cloud}}^{\text{measure}}$ and $E_{\text{eval}}^{\text{measure}}$ are correct.
5. The sender encrypts its input, $\text{input}_{\text{sender}}$, and a public key for its output, $\text{Out}^{+}\text{key}$, under $E_{\text{cloud}}^{+}\text{key}$, to create $\text{Enc}(\text{input}_{\text{sender}}||\text{Out}^{+}\text{key})$ and sends it to the cloud.
6. The cloud enters $\text{Enc}(\text{input}_{\text{sender}}||\text{Out}^{+}\text{key})$ into the SGX program, $\text{prog}_{\text{sgx}}$. We note here that there is no reason the cloud cannot also provide input to the program if desired.
7. The input is sent from the cloud to the evaluator.
8. $\text{prog}_{\text{sgx}}$ is run according to the previous SGX-supported 2P-SFE protocol.
9. The sender’s output, $\text{output}_{\text{sender}}$, is encrypted under $\text{Out}^{-}\text{key}$ as a final step in $\text{prog}_{\text{sgx}}$.
10. This value, $\text{Enc}(\text{output}_{\text{sender}})$, is sent from the evaluator to the sender.
11. The sender uses the output private key $\text{Out}^{-}\text{key}$ to decrypt $\text{Enc}(\text{output}_{\text{sender}})$.

Security of the Sender’s Data

Input: Because the sender’s input is encrypted under the evaluator’s enclave private key, it can only be decrypted inside of the evaluator’s enclave. Given the measurement of the evaluator’s enclave, we also know that the program inside of the enclave is correct; so it will not pass the input outside the enclave.

Output: Because the sender’s output is encrypted inside the enclave during evaluation and is only sent outside when it is encrypted under the sender’s public key, only the sender can decrypt and read this output.
4.4 Improving the security of 2P-SFE protocols using SGX

Semi-honest or honest-but-curious protocols guarantee security as long as all parties faithfully follow the protocol. Such protocols are much cheaper in terms of computational cost than those that protect against malicious adversaries, who attempt to gain additional information by any means necessary. We can use SGX for parts of the semi-honest 2P-SFE protocol to gain additional security guarantees without incurring significant overhead.

First, we replace the OT in the 2P-SFE protocol with an SGX component that acts like an OT. The SGX OT is a stripped-down version of the previously described SGX-supported 2P-SFE protocol. In this program, the 2P-SFE evaluator chooses the encrypted form of the input as in the 2P-SFE protocol. This immediately gives us greater security than the standard semi-honest OT, because we are not relying on the parties to behave correctly during the OT (i.e., the SGX unit checks whether the parties are running the correct “OT” program). Note that this does not guarantee fair release of the result, because a malicious party can still cause us to abort at any point.

Similarly, we can replace the circuit generation and evaluation with an SGX component. This SGX-evaluation is the program-evaluation component described earlier. While we could use the 2P-SFE OT before this part of the protocol, using the SGX OT component gives us better security. After the input and circuit-evaluation components are replaced, we can also replace the output component with the SGX output protocol. Replacing all of these elements leaves us with a protocol that is significantly more secure than the original semi-honest 2P-SFE protocol (because the SGX protocol has checks for when a user is malicious), while remaining much cheaper than a malicious 2P-SFE protocol.

4.5 Universal Programs (Circuits)

A universal circuit (UC) is a program that takes another program as input (denoted as \(UC_{\text{prog}}\)) and then executes it. In a UC for two parties, one party enters \(UC_{\text{prog}}\) as input while the other party enters the input for \(UC_{\text{prog}}\).

However, in 2P-SFE, a UC requires a massive number of array accesses because of the nature of oblivious data access. For each operation in \(UC_{\text{prog}}\) (e.g., \(data[a] = data[b] + data[c]\)), the inputs to the operation (i.e., \(data[b]\) and \(data[c]\)) have to be found from all the possible values that could be entered into the instructions – i.e. this requires a set of if statements to check whether index value \(v\) equals \(b\) – unless constraints can be added to \(UC_{\text{prog}}\). However, in SGX-supported 2P-SFE, this would be more efficient, because array access takes \(O(1)\). Thus, UC programs can be efficiently and privately executed in an enclave.

4.6 Novel Use Cases for SGX

Secure data storage: With the advent of cloud and multi-user systems, unauthorized data access is a greater problem than ever before. Our idea is to use
SGX as a gatekeeper: If all reads and writes went through the SGX hardware, we could automatically encrypt and decrypt it based on a user-entered key without the need for a specialized drive. A keyboard could enter the enclave password while skipping the operating system and any keyloggers within. Unlike systems such as BitLocker [18], the key here would remain safe even if the operating system were compromised. For cloud storage, the SGX program would encrypt data before they are sent to the cloud server; it could be implemented so as to be transparent to the end user and obviate the need to trust cloud companies.

User Authentication: SGX offers many new avenues for user authentication. It includes MRSIGNER, which signs the enclave before it is deployed. Group authentication is also possible, using EPID (Enhanced Privacy ID) [1], an extension to the Direct Anonymous attestation scheme used in [21, 22]. This allows an enclave to sign communications while maintaining privacy within a group. There is also a “pseudonymous” mode, which relaxes the security slightly, allowing the verifier to know whether it has checked an enclave in the past while still maintaining intra-group anonymity.

Cyber-physical applications: Given the security concerns involved in control systems for sensitive infrastructure (e.g., a nuclear power plant or a hydroelectric dam), improving security is highly desirable. In order to prevent attacks on such systems, the controls could be made accessible only through an enclave that would require all orders to the system to be signed; the current state of the system would also be hidden. Periodic signed updates from the enclave to a “master” control system would prevent the system from being taken offline without the knowledge of the master control system. These strategies would mitigate the threat of hackers breaking into the system and altering code or stealing passwords – this information would exist only inside of the enclaves.

Online Games: Online games are played by multiple users on different machines. In order to reduce bandwidth, many games only transfer events, e.g., information for each user command. Each machine can then process events independently but at the cost of each machine’s knowing the entirety of the game’s data, including sensitive information about other players’ positions. SGX could be used to protect private data from other gamers. If each gamer’s private data are inside an enclave, a hacker (or any user who uses a tool to read information normally not available to him) is denied access to private information. The enclave would release such private information to the local machine based on triggers in the code, e.g., when an enemy unit is nearby. We can periodically verify the state of each enclave to prevent cheating.

5 Previous Work on Secure-Execution Environments

In this section, we briefly discuss previous work on the use of specialized software and hardware platforms to enable secure execution of code. None of these works provides the same guarantees or addresses the same scenarios as a 2P-SFE protocol. Various levels of code and data protection have been achieved
using approaches as varied as managed runtime environments (such as Java and .NET), tamper resistant software [3], and microkernels.

Haven [5] is an SGX-based system for executing Windows applications in the cloud. VC3 [42], also based on SGX, allows verifiable and confidential execution of MapReduce jobs in untrusted cloud environments.

Systems such as TrustedDB [4] and Cipherbase [2] use different kinds of trusted hardware to process database queries over encrypted data. There exist several other systems [30,39,47] that use trusted system software (usually a trusted hypervisor) along with specialized hardware to achieve various security and privacy requirements. Some, such as Virtual Ghost [15] and Flicker [34], avoid hypervisors by using specialized kernel-level hardware-isolation mechanisms and time-partitioning between trusted and untrusted operations, respectively. Super-distribution systems for transmission of protected digital data also exist [27]. They decrypt protected data using a key from an authorized clearinghouse and then re-encrypt the data with a locally generated key on the end-user system, ensuring that no one else can use the data. Secure co-processors [44] allow programs to execute securely as long as users can verify that they are dealing with untampered programs and hardware.

Intel has a number of whitepapers on SGX [1,25], as well as previous attempts in the same vein, such as the Trusted Execution Technology [20]. ARM trustzone for Cortex-A processors also provides some similar guarantees and has been used to build embedded linux platforms [49], language runtimes for mobile applications [41], and many other systems.

6 Conclusion

This paper presents the first systematic consideration of Intel’s Software Guard Extensions as a platform on which to implement two-party secure function evaluation. We show that careful use of SGX primitives can facilitate extremely efficient 2P-SFE protocols, provide an outsourcing mechanism for machines without an SGX module, and discuss augmentations to SGX which provide stronger guarantees against leakage. We also use SGX to convert 2P-SFE protocols secure against semi-honest adversaries into ones secure against malicious adversaries, and discuss a number of use cases for SGX. As SGX-enabled processors eventually make their way onto the market, future work will include implementations and improvements to the efficiency and security properties of these protocols.

Acknowledgements: The first author was supported in part by DARPA contract FA8750-13-2-0058. The second and fourth authors were supported in part by NSF grants CNS-1540217 and CNS-1540218. The third author was supported in part by NSF grants CNS-1407454 and CNS-1409599. The fifth author was supported in part by NSF grants CNS-1464087 and CNS-1464088. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, NSF, or the U.S. Government.
References

29. Libicki, M., Tkacheva, O., Feng, C., Hemenway, B.: Ramifications of DARPA’s PROCEED Program. RAND, Santa Monica (2014)
48. Vipindeep, V., Jalote, P.: List of common bugs and programming practices to avoid them (2005)
**CallForFire: A Mission-Critical Cloud-based Application Built Using the Nomad Framework**

Mamadou H. Diallo, Michael August, Roger Hallman, Megan Kline, Henry Au, and Vic Beach

US Department of Defense, SPAWAR Systems Center Pacific
{mamadou.h.diallo, michael.august, roger.hallman, megan.kline, henry.au, vic.beach}@navy.mil

**Abstract.** In this demo paper we describe CallForFire, a GIS-based mission-critical defense application that can be deployed in the cloud. CallForFire enables secure computation of enemy target locations and selection of firing assets. It is built using the Nomad framework, which enables the development of secure cloud-based applications. Our experimental results validate the feasibility of this application within the Nomad framework.

1 INTRODUCTION

Cloud computing provides many benefits for organizations, including improved IT cost efficiencies, scalability, flexibility, and accessibility. However, the confidentiality of data stored within public clouds is not guaranteed due to the multiple cloud security threats identified by the Cloud Security Alliance [2] as well as other surveys [1]. As a result, organizations with sensitive data, especially government agencies, are hesitant to make use of public clouds.

Various technologies have been suggested to address the security concerns associated with storing and processing sensitive data in off-premise public clouds. One such technology is Fully Homomorphic Encryption (FHE), which enables computations to be performed directly on encrypted data. Over the past few years, the cryptographic research community has introduced efficient FHE schemes [5, 9]. The Nomad framework [4] takes advantage of these efficiencies, thereby enabling developers to build applications that leverage FHE for secure computation.

In this demo paper, we describe CallForFire, a prototype cloud-based application that uses the Nomad framework to implement the “Call for Indirect/Supporting Fire” protocol [7]. This protocol is used by infantry to observe and attack enemy targets. Nomad’s underlying storage system is a FHE-based key/value store, which enables storage, computation, and retrieval of encrypted data in the cloud. The use of FHE ensures the confidentiality of the data that is stored and processed in the cloud by mission-critical applications. Nomad uses HElib, an open source FHE library [6] that implements the Brakerski, Gentry, and Vaikuntanathan (BGV) FHE scheme [3]. While CallForFire runs slowly due to the intensive FHE operations, the experimental results show that it is feasible for interactive applications.
2 NOMAD FRAMEWORK OVERVIEW

The Nomad framework provides building blocks for ensuring the confidentiality of the data stored and processed in the cloud by using FHE. It abstracts out the underlying mechanisms for protecting the data so that developers can focus on building the value-added capabilities of their applications. This framework has the benefits of speeding up and simplifying the development of secure applications deployed in the cloud.

Nomad is designed using the client/server architecture paradigm. The design is modular, which enables extensibility and customization of the framework. The architecture of the framework is depicted in Figure 1, which is composed of two main components: the Client Management Service and the Cloud Storage Service. It is assumed that the Client Management Service will be deployed in a trusted infrastructure, and that insider attacks are still a threat. The Cloud Storage Service is assumed to be deployed in a semi-trusted cloud infrastructure. In the following sections, we describe the Client Management Service, the Cloud Storage Service, and the Operational Overview of Nomad.

2.1 Client Management and Storage Services

The Client Management Service provides both the client management and graphical user interfaces for the system. It consists of multiple components (see Figure 1). The Client Management Engine orchestrates the client operations, which include encryption, decryption, homomorphic arithmetic operations, key generation, and key management. The client-side FHE Processing Engine is used for encrypting and decrypting the application’s data. In order to store the data in the FHE-based storage, the following steps are performed: (1) homomorphic encryption is performed on the data using the public key, and (2) the public key is sent along with the ciphertext to the server for storage. Note that the public key is needed by the server-side FHE Processing Engine in order to perform re-encryption (i.e. bootstrapping) and operations on the ciphertext. The Public/Private Key Store persists both the public and private keys associated with each user of the system. The client-side FHE Key Manager is responsible for generating public/private key pairs, storing/retrieving them in the Public/Private Key Store, and encrypting/decrypting data. The Client API is used for exposing these services to...
applications, and the Cloud Monitor GUI is used for monitoring the resource usage of
the virtual machines in the cloud.

The Cloud Storage Service enables the deployment of trusted storage and com-
putation services within a semi-trusted cloud environment. The Cloud Storage Engine
orchestrates all of the operations of the Cloud Storage Service and provides the server-
side interface to the storage system. The cloud service provider’s underlying hypervisor
generates and manages the Virtual Machines (VMs) which host Nomad’s cloud ser-
vices. The Monitor collects the resource usage of each VM periodically and stores it in
the Dom Stats DB (i.e., the domain statistics database). The server-side FHE Process-
ing Engine is used for processing the homomorphically encrypted data, and stores the
results along with the original ciphertexts in the Ciphertext Store. The server-side FHE
Key Manager keeps track of the public keys used to encrypt the data.

2.2 Nomad Operational Overview

The operations performed by the Nomad framework include encryption/decryption,
storage, retrieval, deletion, and processing of the data. The Cloud Storage Service func-
tionality is exposed to the client application via the Client API. The Client Management
Service manages all of the keys, data, and operations on behalf of the client. At a high
level, the client application data is stored and processed in the cloud in encrypted form,
then returned to the client application and decrypted for display to the end-user. When
first using the system, the user must initialize the client and generate their own pub-
lic/private key pair ($key_{public}$, $key_{private}$). Note that, in a deployment environment, key
generation must be done by a trusted third party, as it is done with current certificate
authorities. Alternatively, key management/distribution could be done using homomor-
phic encryption, as is being done by Porticor [10].

Data Storage Workflow. In this data storage workflow, we assume that each user has
a single public/private key pair used for encrypting and decrypting their data.

1. System Initialization: Upon first using the system, the user sends a request to the
Client Management Engine to generate a public/private key tuple

   ($<ID_{user}, key_{public}, key_{private}>$). The Client Management Engine forwards the re-
quest to the HE Key Manager to generate the key pair and store it in the
Public/Private Key Store. The Client Management Engine also sends the tuple

   ($<ID_{user}, key_{public}>$) to the Cloud Storage Engine for later usage. The Cloud Stor-
age Engine calls on the HE Key Manager to store the tuple ($<ID_{user}, key_{public}>$)
in the Public Key Store.

2. The user initiates a request to store their $Data_{plaintext}$ in the cloud storage.

3. The Client Management Engine submits a request to get the ciphertext ($Enc(Data_{plaintext}, key_{public}) = Data_{ciphertext}$).

4. The Client Management Engine submits a request to the Cloud Storage Engine to
store the ID/data tuple ($<ID_{user}, ID_{data}, Data_{ciphertext}>$).

5. The Cloud Storage Engine receives the storage request and calls on the HE Pro-
cessing Engine to store the data ($ID_{user}, <ID_{data}, Data_{ciphertext}>$) in the Ciphertext Store.
3 APPLICATION: CALL FOR FIRE

In this section, we describe the design of the CallForFire prototype. Defense organizations may call for indirect/supporting fire during combat operations when an infantry unit is impractical for engagement with a target. CallForFire places the sensitive computations involved in calling for indirect or supporting fire, specifically computation of a target location, into a secure, homomorphically encrypted cloud environment.

In a tactical environment, the call for indirect/supporting fire procedure involves multiple players. The Forward Observer (FO) observes an adversary target, or “High Value Target” (HVT). They determine the Observer-Target distance ($OT_{distance}$) and bearing ($OT_{direction}$) using technology such as a laser range finder or other means. The FO then homomorphically encrypts the ($OT_{distance}$, $OT_{direction}$) and transmits the information to the “Fire Direction Center” (FDC). Since the FO location is already known, the FDC uses the FO’s position and the ($OT_{distance}$, $OT_{direction}$) to calculate the HVT location. Once calculated, the information is sent to the “Firing Unit” (FU), which “fires for effect” on the HVT. Note that the FDC is not fully trusted due to insider threats, which is the main reason FHE is used to enforce the “need to know” restriction. FHE is also needed to securely outsource the computation to the cloud.

CallForFire uses the Military Grid Reference System (MGRS) [8] rather than the more widely known latitude/longitude coordinate system. The reasons for choosing MGRS are twofold: (1) it is the geocoordinate standard used by all NATO militaries, and (2) all MGRS coordinates are alpha-numeric with letters and integers which are easily handled by all FHE schemes. MGRS coordinates consist of a grid zone designator (GZD)–a double digit integer followed by a letter, a 100,000-meter square identifier (SQID)–two letters, followed by the numerical location (easting and northing) within the 100,000 meter square–both with the same number of digits, which varies from 1 to 5 depending on the MGRS precision/resolution level. The lowest value 1 corresponds to precision level 10 km, while the highest value 5 corresponds to precision level 1 m. The precision level of 1 m is used in the CallForFire application. Given a reference point, distance, and bearing, the numerical location of any position can be computed.

To compute the HVT location in MGRS, it is assumed that the GZD and SQID are known. Also assumed is that the FO location within the 100,000-meter square is known and the HVT is within the same square. Let $FO_{easting}$ and $FO_{northing}$ be the FO’s easting and northing. Similarly, let $HVT_{easting}$ and $HVT_{northing}$ be the HVT’s easting and northing. The $OT_{direction}$, $\theta$, is referenced from 0° north, moving clockwise. Note that the trigonometric function values are pre-computed to four decimal places, appropriately scaled for computation as integers, and then stored in the FHE-based storage. The HVT’s location is calculated as follows:

$$HVT_{easting} = FO_{easting} + OT_{distance} \times \sin(\theta)$$
$$HVT_{northing} = FO_{northing} + OT_{distance} \times \cos(\theta)$$

Initially, CallForFire was tested using only a single FO and HVT [4]. In an actual combat environment, there are likely to be multiple FOs, HVTs, and FU’s, and a single FDC would need to process many calls for indirect/supporting fire simultaneously. Therefore, CallForFire has been expanded to handle multiple FOs requesting indirect/supporting
fire on adversary HVTs, and selectively assigning the HVTs to different FUs based on predefined criteria (i.e., firing asset selection). The firing asset selection process involves computing the distances between all FUs and a given HVT, and selecting which FU to direct fire on the HVT. In a real world scenario, asset selection would consist of more criteria than just the distance between the FU and the HVT. It will be assumed that the positions of the FOs and FUs will be known to the FDC. Multiple FOs may call for fire support on the same HVT. The distance is calculated as follows:

\[
\text{Distance}_{FU-HVT}^2 = (FU_{easting} - HVT_{easting})^2 + (FU_{northing} - HVT_{northing})^2
\]

Figure 2 is a screenshot of the actual CallForFire GUI in a web browser. It shows an example scenario with the following players: 1 FDC, 4 FOS, 5 FUs, and 3 HVTs. In this scenario, the FDC has computed the locations of the three HVTs using the information given by the FOS. It has also selected the nearest FU for each HVT as indicated by the lines between them on the map.

![CallForFire GUI screenshot](image)

CallForFire Operational Workflow. The CallForFire operational workflow describes how the call for indirect/supporting fire procedure is simulated by the CallForFire application. Nomad currently supports the following integer operations in the encrypted domain: addition, subtraction, multiplication, and negation. For this simple scenario, we assume that all locations (except the FDC) are inside of the same MGRS zone.

1. The FO detects an HVT in the field, estimates its distance and bearing, and enters the data into the FO client application.
2. The FO client application uses the FDC public key to homomorphically encrypt the FO’s location (easting and northing) and the HVT’s distance and bearing, and sends them to the FDC.
3. The FDC outsources the computation of the HVT’s location to the Nomad cloud service by sending the homomorphically encrypted FO’s location, the HVT’s bearing and distance, and locations of the available FUs over to the cloud.
4. The cloud homomorphically computes the HVT’s absolute location and selects the nearest FU to direct fire on the HVT.
5. The cloud sends the HVT’s location and FU selection back to the FDC.
6. The FDC decrypts the HVT’s location and the FU selection, then makes the final decision to initiate the firing operation.
7. The FDC encrypts the HVT’s location using the FU public key and sends the firing command to the selected FU.
8. The selected FU decrypts the HVT’s location and directs fire on the HVT.

4 IMPLEMENTATION AND EXPERIMENTS

The Nomad framework is designed to be modular and extensible, using Thrift as the underlying client/server framework. This enables an open architecture, allowing developers to extend the framework, including using different hypervisors for virtual machine management, and choosing different key/value stores for back-end storage. We used Xen as the underlying hypervisor and LevelDB as the key/value store. We implemented the GPGPU-based acceleration technique as described in [4], which uses the Nvidia CUDA programming platform, for a limited number of subroutines. For the Client Management Service, we used the CppCMS web development framework to integrate the different C++ libraries including HElib, Thrift, LevelDB, and Nvidia CUDA. We used OpenLayers as the mapping technology for visualizing the information.

We have extended the CallForFire application, which was originally described in [4]. The extension of the application allows us to examine the performance improvement resulting from HElib batching of operations. In this extension, we have increased the number of Forward Observers (FOs), Firing Units (FUs), and observed targets (HVTs). We compute the locations of all the HVTs using the information given by the FOs. Once the locations of all the HVTs are known, then we identify which FU to assign to each HVT. This determination is based on the distance between the HVT and the FU. We can then compare the performance between batched and non-batched (individual) calculations. We performed experiments to analyze the performance of CallForFire with respect to the overhead associated with Computation, Storage, and Data Transmission. We ignored the latency between browser and server.

HElib uses the following main parameters: R (number of rounds), p (plaintext base), r (lifting), d (degree of the field extension), c (number of columns in the key-switching matrices), k (security parameter), L (number of levels in the modulus chain), s (minimum number of slots), and m (modulus). For all the experiments, the following parameters are fixed: $R = 1$, $r = 1$, $d = 1$, $c = 2$, and $s = 0$. We adjusted the parameters $p$ and $k$ in order to evaluate the performance tradeoffs associated with having a larger integer space and a higher security level, respectively. The parameters $L$ and $m$ are automatically generated by HElib based on the other parameters.

The experiments were performed using two HP Z420 desktops with 16 GB RAM and 500 GB storage, and one MacBook Pro with 2.6 GHz Intel Core i7, 16 GB RAM, and 500 GB storage. The setup is as follows: FO (MacBook), FDC (Z420), Cloud (Z420).
### Table 1. Average Computation Overhead in Sec. with Fixed $p=9576890767$ (10 digits)

<table>
<thead>
<tr>
<th>k: Security parameter</th>
<th>80 ($L=11, m=11021$)</th>
<th>100 ($L=11, m=12403$)</th>
<th>120 ($L=11, m=13019$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Individual</td>
<td>Batched</td>
<td>Individual</td>
</tr>
<tr>
<td>Location Encryption</td>
<td>702.3990</td>
<td>63.0778</td>
<td>782.6890</td>
</tr>
<tr>
<td>Location Decryption</td>
<td>600.7040</td>
<td>165.2790</td>
<td>692.3490</td>
</tr>
<tr>
<td>Location Computation</td>
<td>212.1974</td>
<td>21.3238</td>
<td>221.7478</td>
</tr>
<tr>
<td>Distance Computation</td>
<td>271.2946</td>
<td>26.3864</td>
<td>283.7557</td>
</tr>
<tr>
<td>Storing Location</td>
<td>2.4743</td>
<td>0.2498</td>
<td>2.7999</td>
</tr>
<tr>
<td>Retrieving Location</td>
<td>16.3833</td>
<td>1.5589</td>
<td>18.0937</td>
</tr>
</tbody>
</table>

### Table 2. Average Computation Overhead in Sec. with Fixed $p=1000000000039$ (13 digits)

<table>
<thead>
<tr>
<th>k: Security parameter</th>
<th>80 ($L=12, m=11639$)</th>
<th>100 ($L=12, m=12851$)</th>
<th>120 ($L=12, m=14279$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Individual</td>
<td>Batched</td>
<td>Individual</td>
</tr>
<tr>
<td>Location Encryption</td>
<td>818.4560</td>
<td>72.2017</td>
<td>850.1300</td>
</tr>
<tr>
<td>Location Decryption</td>
<td>696.9510</td>
<td>181.3630</td>
<td>738.4030</td>
</tr>
<tr>
<td>Location Computation</td>
<td>227.4829</td>
<td>23.1525</td>
<td>253.9844</td>
</tr>
<tr>
<td>Distance Computation</td>
<td>287.5192</td>
<td>28.1036</td>
<td>295.7541</td>
</tr>
<tr>
<td>Storing Location</td>
<td>2.5129</td>
<td>0.2454</td>
<td>2.7538</td>
</tr>
<tr>
<td>Retrieving Location</td>
<td>16.5563</td>
<td>1.7040</td>
<td>18.2733</td>
</tr>
</tbody>
</table>

**CallForFire Computation Overhead.** To measure the computation overhead, we performed two sets of computations: (1) calculation of the HVT’s location and, (2) firing asset selection. In the HVT location calculation, we measured the time it took to homomorphically encrypt 10 individual locations consisting of 6 parameters ($GZD$, $SQID$, $FO$easting, $FO$northing, $OT$distance, $OT$direction) each, computed the numerical location (easting and northing) of the HVT for each $FO$, and decrypted the HVT locations. We also measured the time it took to store and retrieve 10 encrypted locations from the storage. In the firing asset selection, we measured the time it took to compute the distance between 10 FUs and an 10 HVTs pairwise. We repeated both experiments 100 times and computed the averages. Table 2 summarizes the results of these experiments and gives a comparison between the performance of individual and batched operations. When performing operations in *batched* mode, an input array with multiple elements is passed in to the storage system. The homomorphic encryption operations can then be performed on all of the elements of the array within the same operation. With *individual* operations, one data element (i.e. an integer) is placed into the input array, which is then passed to the storage system. Based on the results of these experiments, it is best to use *batch* mode when possible, which can reduce the overhead significantly.
Transmission and Storage Overhead. For the transmission and storage overhead, we measured the time it took for the \textit{FO} to encrypt and transmit the location information to the \textit{FDC}, and for the \textit{FDC} to store the information in its database. We considered scenarios for 100 \textit{FOs} and calculated the averages. The time it takes to transmit an encrypted location and store it in the database is about 22 times longer than when the location is not encrypted. For the storage space overhead, the average space used to store a location using HE is 8.96 megabytes, whereas the average for a location without using HE is 17.6 bytes. This significant storage space overhead is a limitation common to all lattice-based homomorphic encryption schemes.

5 CONCLUSION

In this paper, we presented \textit{CallForFire}, a cloud-based mission-critical defense application built using the Nomad framework. \textit{CallForFire} takes advantage of Nomad’s Cloud Storage Service to encrypt and compute enemy target locations in the battlefield. In order to accelerate FHE operations, we investigated the use of GPGPU programming techniques to parallelize some of the HElib subroutines. Our preliminary results show some improvement in the performance of HElib. While the overall performance of HElib may still be impractical for many applications, certain interactive applications, such as \textit{CallForFire}, can still make use of HElib in a limited context to enhance data confidentiality. Further development of HE libraries such as HElib will likely accelerate the adoption of cloud computing by organizations with sensitive data.

References