Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Multi-Point Function Secret Sharing using Cuckoo Hashing

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von:	Leonie Reichert
geboren am:	17.08.1993
geboren in:	Kusterdingen
Gutachter/innen:	Prof. Dr. Björn Scheuermann
	Prof. Dr.-Ing. Eckhard Grass

eingereicht am: .... verteidigt am: ....

Selbständigkeitserklärung

Berlin, den 16. August 2019 ....

Contents

Abstract

Function secret sharing (FSS) is a useful primitive for secure multi-party computation. It creates secret shares of functions over a domain with size n. In this two party protocol, each party receives a vector as final result. Combining these vectors gives the input function evaluated over $[0, n]$. So far, FSS implementations have focused on distributed point functions (DPF), which is the secret sharing of point functions. A point function is unequal to zero at exactly one location.

More complex is the creation of secret shares of multi-point functions, also called multi-point function secret sharing (MPFSS). MPFSS shares for example can be used as source of correlated randomness in vector oblivious linear-function evaluation (VOLE). A straightforward method to accomplish MPFSS is to run a fixed amount of DPFs over the same domain and then combine the results.

Boyle et al. [9] proposed a technique optimizing the creation of MPFSS through batching. Instead of the simple approach, the domain is split into buckets with lengths significantly shorter than n. Indices of the multi-point function are assigned to buckets. Then one DPF per bucket is called. To solve the assignment problem, the authors leverage the primitive of combinatorial batch codes. The basic idea of this optimization is saving costs by having more DPF runs on shorter domains. While the authors give estimates, they did not implement their protocol and evaluate its real world trade off. This work focuses on a similar optimization with the main difference that instead of applying combinatorial batch codes, cuckoo hashing is used. Cuckoo hashing allows constant look-up time and sublinear insertion time even in the worst-case. A lot of research has been done applying cuckoo hashing to MPC and many works have employed it for their purposes. Here, it is used in the configuration with three hash functions, random-walk selection and no stash.

The MPFSS protocol proposed on the basis cuckoo hashing was implemented and evaluated. The analysis shows that it performs significantly better than the naive approach for almost all parameters. It is faster especially for large values of domain size and number of indices, which are relevant in practice.

1 Introduction

Secure multi-party computation (MPC) deals with creating protocols for joint computation on private, distributed data without a central instance. This is for example useful for hospitals or companies working on sensitive data. MPC algorithms often rely on correlated randomness to hide private information. One approach to correlated randomness is to give each party participating in the MPC one seemingly random vector, called share. Combining all shares would yield a meaningful result. Using this, it is possible for each party to conceal a private vector with their random share. The result can then be used in a joint computation. Shares can later be used to obtain the true value of the final result.

Creation of such shares for a two-party MPC protocol can be realized using function secret sharing of (distributed) point functions (DPF). A distributed point function f is shared between both parties in a way that each party only holds a secret share of length n, which each for themselves hide f. Combining these shares will yield f evaluated on the domain $[0, n]$. The function is defined as being zero everywhere except for one specific position at which it has a fixed amplitude. Analogously, it is possible to do multi-point function secret sharing (MPFSS). Similar to a distributed point function, a multi-point function g has t indices with an amplitude value unequal to zero. MPFSS is therefore a method for creating two pseudorandom vectors, which combined yield the multi-point function g.

MPFSS can for example be utilized to implement *vector oblivious linear-function evaluation* (VOLE) [9], a method of realizing vector-scalar multiplications in a secure, distributed manner. A set of such vector-scalar multiplications can be used to realize matrix multiplication, which is essential for machine learning. This means that creating efficient MPFSS is a step towards feasible secure machine learning. Another application is *oblivious random access memory* (ORAM). Here, performance of an existing protocol can be improved by combining multiple instances of DPF into a single run of MPFSS.

Construction methods and implementations of DPF already exists. It can therefore be used as building block for MPFSS. A simple way to do MPFSS would be to call a DPF t-times on the whole domain n. Since a single DPF is in $\mathcal{O}(t)$ it requires a runtime of $\mathcal{O}(nt)$. A more promising approach is to leverage batching as it allows to decrease domain size for single DPF runs. Here, the domain is spit into m buckets ($m \geq t$). All indices will be assigned to buckets and a single DPF is called for each bucket. Similar to other batching mechanisms, elements from the domain have to be inserted into multiple buckets. The method ensures that for any set of t indices there exists a set of bucket so each index can be assigned to a bucket.The number of elements counted when combining all buckets is larger than n. But since each DPF is run on a smaller input domain (the size of a single bucket instead of n), performance improves for some combination of parameters.

The contributions of this work are summed up in the following paragraph. This thesis explores the practical performance impact of batching for MPFSS and the use of *cuckoo hashing* to facilitate finding assignments from indices to buckets. The approach is therefore called MPFSS Cuckoo or MPFSS using cuckoo hashing. Cuckoo hashing allows worst-case constant look-up time and sublinear insertion time. It is used with three hash functions so no stash is necessary and table utilization is high. The evaluations of Angel et al. [3], shows that cuckoo hashing can be tuned to fail with negligible probability. To be able to employ it in this context, the definition of MPFSS had to be relaxed. This relaxation requires one party to know all indices of the multi-point function. Unlike the batching methods proposed by Boyle et al., this algorithm is implemented and evaluated. The evaluation compares a naive approach with MPFSS using cuckoo hashing. It shows that the latter is significantly faster for almost all parameter setting in the LAN and over the Internet. Only for very low values of t and n, the naive approach performs better.

The remaining work is structured as follows. In Section 2, fundamental terms and constructs are introduced. Section 3 discusses the use of cuckoo hashing in MPC and expands on the approach of Boyle et al. The MPFSS Cuckoo algorithm and its applications are described in Section 4, while the actual implementation and design decisions are explained in Section 5. The algorithm is evaluated and compared in Section 6. Discussion follows in Section 7.

2 Background

2.1 Secure Multi Party Computation

Secure multi-party computation (MCP) [33, Chapter 22] deals with creating protocols for joint computation on private, distributed data. It studies mechanisms to allow a group of n interdependent participants to collectively evaluate a function $y_1, \cdots, y_n = f(x_1, \cdots, x_n)$. Each participant holds a secret x_i, which shall remain hidden. The participants only learn their final result y_i, but not the input data of others. Any function f, that is solvable in polynomial time can be represented as MPC protocol [33, Chapter 22.2].
Depending on the function, it is possible to deduce some information for the output. Let's take a look at two parties P_1 and P_2 who want to calculate a result for the following term:

$$((a \text{ AND } b) \text{ OR } (c \text{ AND } d))$$

Party P_1 holds a, b. P_2's inputs are c, d. Knowing the result and its own input, P_2 is able to deduce the value of (a AND b), as the function itself leaks information.
MPC protocols run in a distributed manner, so there is no need for the participants to trust each other or a central instance.

2.2 Models for MPC

One way trivial to realize a protocol for multiple parties would be to send all their inputs to a central server which computes the results. In the best case, the center instance is trustworthy. This assumption is difficult to prove in practice, but represents the ideal setting for such a protocol. In MPC the execution using such a trusted third party is defined secure [17, Chapter 7.1]. A MPC scheme is considered secure, if all attacks against the real world protocol are also possible in an ideal setting. This means one or multiple adversaries participating in an actual execution would have the same possibilities in a hypothetical, ideal protocol with similar functionality. In such a case, the scheme successfully emulates an ideal situation. This implies, that honest parties will receive a correct outputs and their inputs remain private (not accounting for the leak resulting from function design).

When trying to keep information secret it is always important to consider the type of attacker to defend against. Hereby the attacker's amount of available resources and willingness to actively alter the flow of data play important roles. Typically, only attackers with limited computational power are considered. These adversaries are called computationally bounded. An example for this are *probabilistic polynomial-time* (PPT) adversaries, which terminate after polynomial amount of operations. Results from a computational unbounded scenario can be transfered to the bounded case.
It is also possible to distinguish adversaries using their behavior:

- *Honest-but-curious*: This case is also called *semi-honest* or passive. The adversary

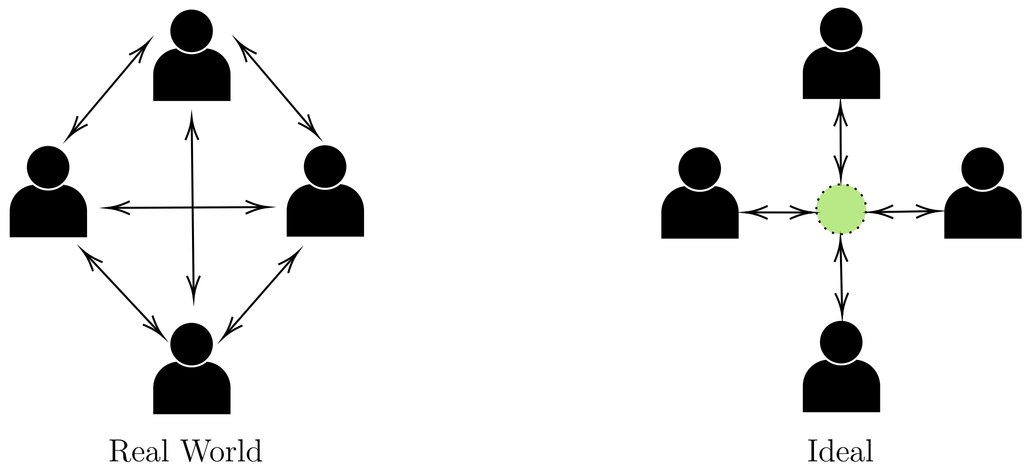


Figure 1: In an ideal setting, a third party exists, that is trusted by all participants. This case is considered to be the secure baseline.

is expected to follow the protocol while passively observing the process and gathering information. Multiple parties are allowed to collude to deduce secrets.

- *Malicious*: Also called active. This attacker can change the flow of information by inserting, altering or dropping data and deviate from the protocol. Again, corrupt parties are allowed to work in coalition. Malicious participants of a protocol can be forced to behave semi-honest [17, Chapter 7.1.3].

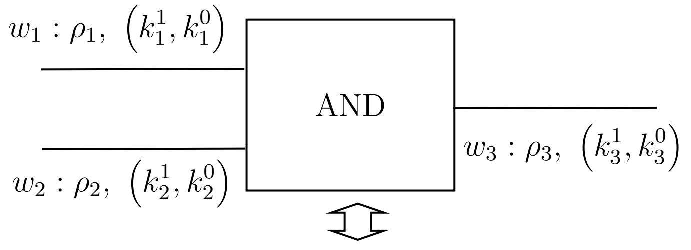
It is generally assumed, that any adversary is capable of tapping connections between all parties, so the encryption of a communication channel itself does not provide privacy. In some settings postulating a private channel between parties can be useful. The adversary is then often considered to be computationally unbounded.
In MPC protocols there is always one party, which is the first to receive their final output. If this party does not follow the protocol, it can block all others from receiving their results by sending an error message. But it will still only learn the same information as it would have if the execution was not suspended. If multiple protocols or runs are executed consecutively, one possibility is to change the party finishing first every time. In the semi-honest model, this is not a concern as parties are expected to follow the protocol.

2.3 Yao Garbled Circuits

One way to realize a MPC protocol are *Yao's garbled circuits* [33]. While there are methods, this approach is best suited for scenarios with two participants. Let $f(x_1, x_2)$ be the function of a two party protocol where x_1, x_2 represent inputs of the respective parties. Function f can be represented as a binary circuit with additional encryption, also called *garbled circuit*. If f is solvable in polynomial time, there exists a polynomial-sized binary circuit yielding the same solution. One party, the garbler, creates a

Boolean circuit from function f, then encrypts it to produce a garbled circuit. Next, this is send to the other party for evaluation. This party is sometimes called evaluator.

Each Boolean circuit consists as a set of logic gates connected with wires. Multiple input wires may exist, but only one output. Each gate can be represented by a truth table, which provides one output value for all combinations of inputs. In garbled circuits, the output is encrypted in a way that actual values can only be retrieved with additional knowledge. The garbler withholds some information from the evaluator so it can not learn intermediate values in clear text but only the final result.


$w_1 : \rho_1, \left(k_1^1, k_1^0\right)$
AND
$w_2 : \rho_2, \left(k_2^1, k_2^0\right)$
$w_3 : \rho_3, \left(k_3^1, k_3^0\right)$

Encrpted Truth Table

e_1	e_2	$c_3^{e_1,e_2}$
0	0	$c_3^{0,0}$
1	0	$c_3^{1,0}$
0	1	$c_3^{0,1}$
1	1	$c_3^{1,1}$

Figure 2: Visualization of a AND gate as Yao garbled circuit from the view of the garbler. Each wire has one ρ value and a pair of keys assigned to it. The encrypted truth table holds all possible outputs in hidden form. Inputs v_1 and v_2 are obscured to e_1 and e_2 by using the respective ρ values.

A garbled circuit is constructed the following way:

1. Select two random cryptographic keys k_i^0, k_i^1 per wire w_i. One key represents the encryption of a zero value. The other serves as placeholder for a one.

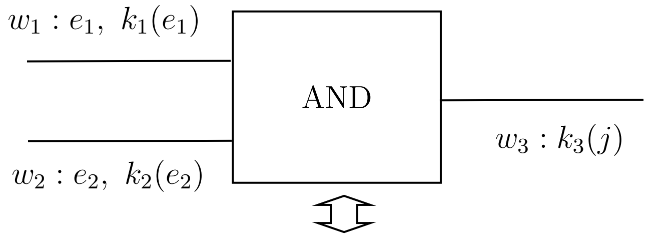
2. Next, a random value $\rho_i \in 0, 1$ is selected for each wire to hide its actual value v_i. These are used to calculate an external value $e_i = \rho_i \cdot v_i$.

3. For each gate, an encrypted truth table is calculated. It holds a list of output values based on combinations of external values. This explanation only focus on gates with two input and one output. Truth table entries for all possible combinations of external values e_1 and e_2 are calculated the following way. The gate functionality g (e.g AND, OR, XOR, NOR, ...) is executed on the actual values $v_1 = e_1 \oplus \rho_1$ and $v_2 = e_2 \oplus \rho_2$ by reconstructing them from the external values. The result, e.g. (v_1 AND v_2), is then XORed with ρ_3, the encryption

value of the output wire. The outcome of this operation is called $j \in 0, 1$ and is represented by key $k_3(j)$. This key is then encrypted using a given secure encryption function E and two other keys. These other keys are $k_1(e_1)$ and $k_2(e_2)$. Each represents an external value e_i as either key k_i^0 or key k_i^1. The result of the encryption is a cipher text $c_3^{e_1,e_2} = E_{k(e_1),k(e_2)}(k_3(j))$. The encryption can be reversed when both keys are known.
A more compact description of this calculation is:

$$\begin{aligned}
e_i &= \rho_i \cdot v_i \text{ for wire } w_i \\
k_i^{v_i} &= k_i(e_i) = \begin{cases} k_i^1, & \text{if } v_i = 1 \\ k_i^0, & \text{if } v_i = 0 \end{cases} \\
j &= g(v_1, v_2) \oplus \rho_3 \text{ for Boolean function } g \\
c_3^{e_1,e_2} &= E_{k(e_1),k(e_2)}(k_3(j))
\end{aligned}$$

Let party P_1 be the garbler in this case. Therefore, party P_2 takes on the role of evaluator. Both parties have a different view of the circuit. While P_1 sees the circuit as shown in Figure 2, P_2's view is presented in Figure 3.



Encrpted Truth Table

e_1	e_2	$c_3^{e_1,e_2}$
0	0	$c_3^{0,0}$
1	0	$c_3^{1,0}$
0	1	$c_3^{0,1}$
1	1	$c_3^{1,1}$

Figure 3: Visualization of a AND gate as Yao garbled circuit from the view of the evaluator. The external values and keys for some input wires will be provided by the garbler. All others external values and keys are established by using one OT per wire. The key $k_3(j)$ is deduced by correctly encrypting the right value of the encrypted truth table. To learn the final result, the evaluator needs ρ_3.

Lets take a look on how party P_2 will deduce the result of a garbled circuit consisting

of a single gate. In this example, input for wire w_1 comes from party P_1. Similarly, for the real value on w_2 comes from P_2. To be able to start, P_2 needs to know key k_i and external value e_i for all input wires w_i of the circuit. From party P_1 it receives the truth table, as well as external value e_1 of wire w_1 and corresponding key $k_1(e_1)$. These hide the actual value v_1 of the wire. To learn external value e_2 and key $k_2(e_2)$ for wire w_2, P_2 uses its own knowledge and calculates $e_2 = v_2 \oplus \rho_2$. It then starts an *oblivious transfer* (OT) [30] with P_1. An oblivious transfer is a basic MPC protocol allowing a party to learn one of two secrets held by the other participant. This side does not learn which secret was selected. So in our case, P_1's secrets are the two keys k_2^0 and k_2^1. Using the OT, P_2 learns the key at index e_2, which will be called $k_2(e_2)$. Knowing e_1 and e_2, P_2 can look up the encrypted value $c_3^{e_1,e_2}$. The keys $k_1(e_1)$ and $k_2(e_2)$ are then used to decrypt cipher text $c_3^{e_1,e_2} = E_{k(e_1),k(e_2)}(k_3(j))$. Party P_2 learns $k_3(j)$ but does not know what value represents, if P_1 does not provide ρ_3. Here it becomes apparent, why the OT is necessary. If P_2 would know more than just the actual keys it could start decrypting multiple values from the truth table.
The value $k_3(j)$ can be used as input for other gates which allows the creation of more complex circuits. P_2 needs to know ρ for the last output wire to retrieve the final result. Since it is not supposed to learn the outcome of intermediate steps, P_1 does not share ρ values for output wires of intermediate gates. P_1 can learn the final result from P_2.

2.4 Secret Sharing

Secret sharing [32] is an essential primitive of MPC. While it is generally defined for arbitrary numbers, the definition is adapted here to only two secrets. A secret s is split into two shares, in a way such that neither reveals further information about s. Only through the combination of both shares it can be reconstructed. Figure 4 explains the concept graphically. In an ideal model, shares are computed by a central trusted party. Therefore, there exits an MPC protocol which enables computation of shares in a distributed manner.

2.4.1 Function Secret Sharing

For *function secret sharing* (FSS)[7] the secret s to be shared is a function f which is split into multiple partial functions. Again, the original definition is adapted to the two-party case. Function f is defined on a domain with size n and can be represented as a vector of length n. The partial function are constructed so that for each x in the domain $f(x) = f_1(x) + f_2(x)$. No f_i by itself allows reconstruction of the original function. Partial functions can be evaluated locally for any x without additional communication overhead.
A two-party FSS scheme consists of two probabilistic polynomial time (PPT) algorithms *Gen* and *Eval* [7]:

- $(k_1, k_2) \leftarrow FSS.Gen(1^\lambda, f)$: Creates a set of keys using f and a security parameter λ. λ governs the computation power needed by a semi-honest attacker to break security.

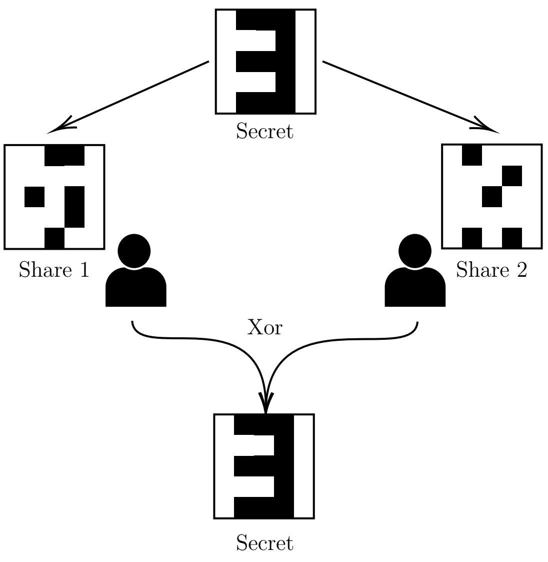


Figure 4: A secret is shared between two parties and can only be reconstructed by XORing both shares.

- $y_i \leftarrow FSS.Eval(i, k_i, x)$: Evaluates for a given key k_i, which hides the partial function f_i, and an input x the function $y_i = f_i(x)$. Index i identifies the party and is defined as $i \in 1, 2$.

After the generation step, keys are distributed to the parties so that each party holds a single key. The evaluation process does not require any communication.
Correctness of am FSS schema is given, when $f(x)$, the function used by $FSS.Gen$ to create a set of keys, is identical to the function derived by combining the output of $FSS.Eval$ for both keys.
The security requirement of FSS demands keys created by *Gen* for two different functions f and g to be computationally indistinguishable by a hypothetical PPT adversary, if they have the same domain.
A benefit of FSS is the fact, that distributing the keys produced by *Gen* and expanding them locally can be significantly faster than simply sending the requested output vectors.

2.4.2 Distributed Point Function

A variant of FSS is the two-party *distributed point function* (DPF) [16, 8]. A point function is defined as

$$g_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$$

DPF allows the representation of $g_{\alpha,\beta}(x)$ as two keys. Each key by itself hides index α and amplitude β.

Gilboa and Ishai [16] proposed an DPF construction enabling the compression of keys to a size sublinear in n, which was later improved by Boyle et al. [8, 7].

A DPF scheme can be created using the tree-based construction proposed in [7]. *Gen* creates two keys k_1, k_2 for a fix α and β, which are distributed to the parties.

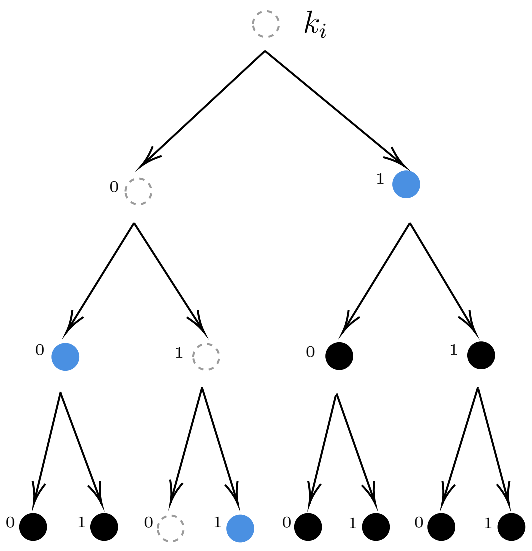


Figure 5: This is the tree of party i created trough *Eval*. The gray dashed nodes are different between the two parties, as they are the path from root k_i to the selected index. All keys at on one level will be corrected jointly once after being created. All blue nodes were influenced by the correction step and are identical in P_1 and P_2's trees afterwards. Black nodes are the same as their counterpart even without correction and are not affected by it.

Each side calls *Eval* with their respective key k_i. This action creates a binary tree. All nodes, expect the ones located on the path from α to the root are identical in P_1's and P_2's tree. Neither party is allowed to know, which nodes differ, as it would reveal index α.

In context of tree construction, keys are also often called "seeds". Using a pseudo-random generator, the initial seeds k_1, k_2 can be expanded into two children each. These children form the next level and can be extended as well. The expansion process terminates once the tree has n leaf nodes.
Let's again take a look at expansion of the initial seed. Since $k_1 \neq k_2$, the product of their expansion will also differ. By applying a correction using additional, layer-specific parameters, one seed on the second level of each tree will be fixed. The two seeds are identical after this step. Calculation of corresponding correction values is complex and therefore out of scope of this work [14]. Both sides apply the correction without knowing which branch will be influenced by it. A correction is applied once per layer on all nodes in the layer. Figure 5 visualizes the fully-expanded tree of as single party for a DPF of size 8.

Since each DPF execution comes with the construction of a binary tree per party, setup costs have to be expected.

2.4.3 Multi-Point Function Secret Sharing

An extension of single point FSS for DPF is *multi-point FSS* (MPFSS). For input domain $[0, n]$ and any $i \in [1, t]$, multi-point function g is defined as:

$$g_t(x) = \begin{cases} \beta_i & \text{if } x = \alpha_i \\ 0 & \text{otherwise} \end{cases}$$

Shared secret here are the indices where multi-point function g is not equal zero and the amplitude values (β values) at these locations. After termination of the protocol, each party holds a vector of length n which represent secret shares of g. This means g can be reconstructed if both shares are combined. The MPFSS definition can be relaxed by for example not requiring indices and amplitude values to be secret shared, but instead to be known to one party.

A naive way of realizing MPFSS would be to run a DPF t times over domain n. Due to the runtime of a single DPF this algorithm take $\mathcal{O}(nt)$. The results of all runs then have to be combined.

2.5 Cuckoo Hashing

Cuckoo hashing[26, 15] is an efficient hashing algorithm with worst-case constant lookup time and sublinear insertion time. Additionally, it allows a high density of items in the table when three or more hash functions are used.

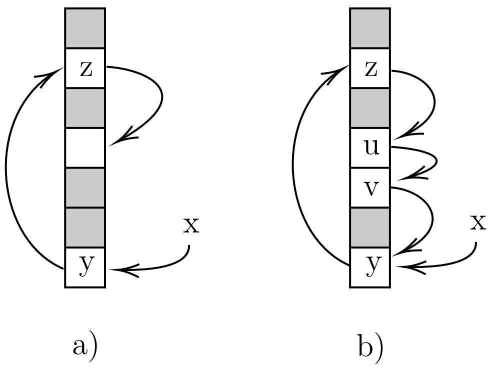


Figure 6: In this figure, cuckoo hashing with two hash functions and one bucket is illustrated. Arrows represent the insertion of a key. On the left side the operation was successful. On the right side a loop in the graph of evictions formed. Here, the $MaxLoop$ bound will ensure resolution.(Figure from [26])

The algorithm uses w hash function $h_1, \cdots, h_w \in H$ to hash any key from the universe U into m tables. Locations in a table are capable of holding only one element. In this work only the configuration using a single table is considered. To insert a key x the first time, h_1 is used to select a location in t. If a collision occurs and location $h_1(x)$ is already in use, the occupying key is evicted and replaced by the new one (see Figure

6). The nestless key will then be reinserted using a hash functions with which it has not been hashed with before. This can potentially cause other keys to be moved. To prevent never ending relocation due to circles in the graph of evictions, the number of iterations is limited by an upper bound also called $MaxLoop$. If this bound is reached, the authors of the original paper [26] propose to select an additional hash function. When the key and all affected ones are successfully inserted, the iteration count can be reset. This process continues until all keys are placed.
It is not always feasible to simply select a new hash function, because it leaks the information that a hashing failure occurred. For this reason reaching $MaxLoop$ is treated as failure in the context of this work.

Hash functions have to be picked carefully, so the development of closed loops is not consolidated trough unwanted interdependence. Following [26], for $w = 2$, all hash functions have to be selected uniformly at random from a $(log(1)), log(n)))$-universal family H. Let's take a random $h_r \in H$, which hashes elements from U to R ($h_r : U \rightarrow R$). Also, let $x_1, \cdots, x_k \in U$ and $y_1, \cdots, y_k \in R$. From the definition of (c, k)-universality [10], the hashes of any k distinct elements $x_1, \cdots, x_k$ do not collide with a probability of:

$$Pr[h_r(x_1) = y_1, \cdots, h_r(x_k) = y_k] < c/R^k$$

This analysis falls apart for larger numbers of w. Fotakis et al. [15] instead propose the use of hash functions with truly random output, meaning they are independent and uniformly distribute keys over the table.

There are multiple ways to select the hash function for rehashing. First proposed [26] was the breadth-first approach, where there is a defined order. Each time a certain key is evicted from $h_i(x)$ it will be reinserted using the subsequent function h_{i+1}. If the location was $h_w(x)$ the function h_1 will be used. Another option is random-walk selection [15]. Here, the keys will be randomly inserted into one of their w locations. Only when $w \geq 3$, this method is a useful alternative. It was first used to minimized the necessary book keeping associated with the breadth-first approach, but was found to have implications on the failure probability and the insertion time. Fountoulakis et al. found that with $w = 3$ and the random-walk heuristic, insertion succeeds with high probability of $p = 1 - \mathcal{O}(1)$ in polylogarithmic time as long as the number of inserted elements does not exceed a certain load threshold.

Theoretical results like these have been criticized [15, 11, 28] as making it difficult to work out actual failure probabilities, since the Landau notation hides constants. Pinkas et al.[28] therefore attempted to empirically determine such a probability for the case of two hash function and a stash. Trough selection of good parameters Chen et al. [11] were able to achieve a failure probability of 2^{-40}. They showed that by choosing $w > 3$, only minimal improvements can be achieved which are canceled out by additional costs. Demmler et al. [13] gave a precise, empirically determined formula for relations between n and the cuckoo table expansion e for $w = 3$. The authors of [15] researched the relationship between memory utilization and number of hash functions and found that for larger w the utilization approximates 100%.

To deal with hashing failure an additional auxiliary data structure called stash [21] was invented. This stash stores items that could not be allocated using normal cuckoo hashing. To guarantee a reasonable failure probability, the stash size has to be logarithmic in the number of items to be inserted. Practical works [11, 13, 3] have shown that for certain parameter choices the stash is effectively not used and can therefore be ignored. It has been demonstrated by [11, 13], that for $w = 3$ cuckoo hashing performs well without stash, while for $w = 2$ it does not. As mentioned earlier little can be gained from choosing more than three hash functions.

3 Related Work

3.1 Preliminaries

In the following chapters parameters and symbols are define as:

- α values: Locations where multi-point function does not evaluate to zero; also sometimes called indices
- β values: List of amplitude values of multi-point function f. For each element in the list: $f(\alpha_i) = \beta_i$.
- n: Size of domain
- t: Number of α values/indices
- d: Duplication factor for batch codes; number of bucket an element $i \in [0, n]$ occurs in
- w: Number of hash functions used for cuckoo hashing
- m: Number of buckets
- $\mathbf{a}$: List of α values, held by party P_1
- $[b]_x$: List of secret shared β values held by party x
- $\oplus$: XOR function

3.2 Cuckoo Hashing in MPC

Cuckoo hashing has many applications in MPC. For example Kolesnikov et al.[22] used the technique to optimize *private set intersection* (PSI). PSI deals with two parties each holding a set of items and wanting to learn the intersection of both sets without revealing their entire content the other side. Let party P_1 hold a set X of size n. The approach of Kolesnikov et al. has P_1 place all elements of its set in a hash table of size $1.2n$ using cuckoo hashing with three hash functions. It additionally uses a stash of size s. Empty slots in the table and on the stash are filled with dummy elements. Both parties then run $1.2n + s$ instances of an *oblivious pseudorandom function* (OPRF). For a single instance of OPRF party P_2 provides a seed S and P_1 learns the output of a pseudorandom function over S and its input r. This way P_1 learns $1.2n + s$ new values. Party P_2 computes OPRF values for the hashes of all its elements with the same hash functions. It then sends them to P_1, which will compute the intersection.

Pinkas et al. [27] also use cuckoo hashing for PSI but to facilitate batching. Both parties each hash their items into a number of buckets and then determine the intersection per bucket. All buckets must be padded to have a predefined length so that no information is leaked. To apply cuckoo hashing, one side (preferably the one with the smaller set) must insert its elements into all possible locations defined by a number of hash functions. The authors experimentally evaluate the influence of stash size, the number of hash functions and the number of buckets on the failure probability. They found that to eliminate the stash while maintaining the same failure probability, it is necessary to increase the number of buckets. Their exact parameter choice largely depends on the ratio between the cardinality of both set.

Chen et al.[11] use cuckoo hashing in a very similar context and experiment with larger number of hash functions. Due to its incompatibility with the selected encryption scheme, they omit the stash. A random-walk based algorithm with three hash function is used. The authors also provide data on the necessary size of the hash table so that cuckoo hashing with these parameters only fails with a probability of 2^{-40}.

Angel at al. adopted cuckoo hashing to *private information retrieval* (PIR)[3]. PIR seeks to hide a users access to a server's database. Users shall be able to access data while the server is unable to learn which data users were interested in. This is relevant, when encrypted data is stored on an untrusted server. Such database interactions always have to read or alter all items in the database to hide the index of the targeted item. It can therefore be very expensive for the server. An approach to improve performance it to split the database into several buckets and distribute these to a set of non-colluding servers. The mechanism proposed for creating splits is similar to the combinatorial batch codes of Boyle et al. in [9].

Demmler et al. [13] use cuckoo hashing and DPFs to calculate the intersection of two concealed, asymmetric data sets. The authors describe a method to search for personal contacts in list owned by a central provider (e.g. messaging service). Contents are protected from crawling, entries not located in the intersection remain private. They consider a setup with one client (holding a local data set X) and two non-colluding servers (each holding a copy of the provider's database DB). A hash table size is selected to ensure that insertion of all elements from DB only fails with a small probability. The two server then insert all element into the table. Based on the size of X, a number of bins m and their length μ is determined. There has to be at least enough space to insert all elements of X with all hash functions once. The servers split DB into m almost equally sized regions. The client creates m empty bins and hashes all "database queries" $x \in X$ into them using each hash functions once per query. As some bins might have different length now, the client pads them to all have the size μ. For each element in each bin m_i, the client generates two keys and sends one key to each server. The servers expand their respective key and calculate its product with DB_i. So there are μ DPF evaluations per bin, each with the domain size μ. To stop the client from learning at which location of the cuckoo hash table one of their contacts is located (there are as many possible positions as there are hash functions), the authors created an oblivious shuffle mechanism to obscure this information. After some shuffling and masking, one designated server learns an intermediate result which it will use to help the client calculate its intersection with DB.
The authors utilize cuckoo hashing to create buckets (from a database) and give the client an idea of where certain elements are located in the table (if they exit). A simple batching mechanism is also applied based on the assumption, that queries of

the client are well spread out across DB. Database and queries are split into bins. The client makes more DPF runs than without batching, but since each run uses a smaller instance (a bin instead of a whole database) performance actually improves.

3.3 MPFSS with Batch Codes

Boyle et al. [9] proposed an optimization running, according to their calculations, in $\mathcal{O}(n)$. To achieve this, they leverage the idea of *combinatorial batch codes*. Here, a number of n items is distributed into m buckets in a way such that to reconstruct a random subset of length t, each bucket has to be accessed only once. To fulfill this requirement, $t \leq m$ and duplication of items is allowed.

Transferring the idea to MPFSS means buckets have to be created and each of the t predefined indices of the multi-point function is assigned to a single bucket. All indices are matched only once. Finding such an assignment can also be considered as matching problem in a bipartite graph (see Figure 7). The t values and the buckets each represented by a set of nodes. Each connection illustrates the occurrence of an index α in a bucket. Therefore, they can not exist between nodes of the same set. An maximal matching is required, since all indices have to be assigned. If one is found, a DPF is executed for each bucket.

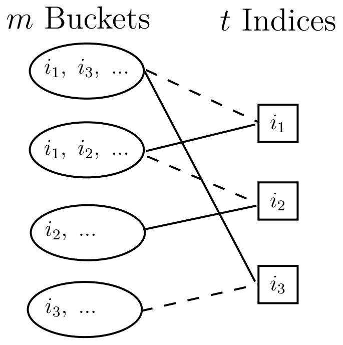


Figure 7: Finding assignment from indices to buckets. Each line indicates that an index α exists in a bucket. Solid lines are part of a solution.

Results can then be combined into one MPFSS secret share per party. A way to create buckets proposed by Boyle et al. was to simply place each $i \in [0, n]$ in d random buckets. With a probability of $p = 1 - t^{-2(d-1)}$, a valid matching exists for a graph created with this method. Therefore, it is possible to maximize this probability, or analogously to minimize the failure rate, by altering d and t. Since m depends on t it will adapt accordingly. Due to their tree-based creation each DPF comes with setup costs. It should therefore be a main goal to keep the number of executed DPFs, which is equivalent to the number of buckets, as low as possible.

So far it was assumed if an assignment exist, it will be found. The following paragraph will discuss this in more detail. The field of bipartite matching has been widely studied in the past and provides a broad range of solutions. For example flow based approaches perform well [34] for the problem at hand. A source is connected to all indices of the graph in Figure 7 and a sink to the buckets. Each edge is given the capacity of 1. Then the flow from sink to source is maximized and if its values reaches t, a valid maximal mapping has been found. Runtime of matching algorithms generally depend only on two parameters: The number of vertices V and the number of edges E. In this case, $V = m + t = t^{1+\epsilon} + t$ and $E = t \cdot d$. The goal is therefore to find a solution running fast with respect to t.

Candidates are the Edmunds-Karp algorithm [34] with a runtime of $\mathcal{O}(V \cdot E^2)$, the Goldberg-Tarjan algorithm [34] with $\mathcal{O}(E \cdot V \cdot log(V^2/E))$ and the Hopcoft-Karp algorithm [19] with $\mathcal{O}((E + V) \cdot \sqrt{V})$. Various sources give $\mathcal{O}(E \cdot \sqrt{V})$ as runtime of the last one [34]. A comparison shows, that the Hopcroft-Karp approach is best suited for the case of batch codes.

Setup costs that can not be amortized were considered essential. Acknowledging necessary intermediate step such as finding an assignment and the combination of DPF output to create meaningful results, gives a runtime is larger than $\mathcal{O}(n)$. Instead, the it can be better approximated with:

$$\mathcal{O}((t \cdot d + t + m) \cdot \sqrt{t + m} + t \cdot \frac{d \cdot n}{m} + n \cdot d)$$

But due to the use of Landau notation, this runtime estimate suffers the problem of large hidden constants and setups costs, which might cause the naive variant to be faster in practice.

4 MPFSS using Cuckoo Hashing

4.1 Basis

Batching as discussed in Section 3.3 is still a promising method for MPFSS but a more efficient mechanisms for finding assignment from buckets to indices is needed.

A better method seems to be the creation of buckets with the assignment task in mind. There are two approaches for this. One option is to create buckets from indices. Each index is the basis for one bucket. The buckets are then padded with elements from the domain in a way that each $x \in [0, n]$ appears d times over all buckets. One main problem of this approach is that buckets can not be reused. New buckets would have to be created for each MPFSS execution. This takes $\mathcal{O}(n)$ every time. The other option is to create buckets in a way, that makes it simple to find an assignment for indices afterwards. Buckets can be reused and costs for their creation can be amortized. Hashing quickly comes to mind as it is known to allow reliable, fast data placement and retrieval. There are many different types of hashing algorithms, but the algorithm required for the task at hand has to fulfill some requirements:

1. Hashing results from the assignment step (executed on a subset of the data) have to mirror the output of bucket creation. An element can only be successfully

assigned, if it actually appears in the bucket to which it was matched. Therefore hashing techniques, where collision resolution depends on the hash table state, cannot be used.

2. The algorithm is also required to spread out elements randomly over all buckets. This ensures that all possible combinations of indices are feasible and none can be excluded beforehand.

3. Another requirement is constant lookup time. It is important that there is only a constant number of locations, where an element can be located.

This reduces the list of candidate algorithms and leaves one with only data-independent hashing methods based on random hashing. Many works have proven the suitability of cuckoo hashing for MPC (see Section 3.2). Angle et al. [3] examined multiple batching methods. Among other things they compared combinatorial batch codes, as used by Boyle et al. [9], with cuckoo hashing and two-choice hashing. They found that the approach using cuckoo hashing yields a smaller number of buckets m and is therefore more efficient.

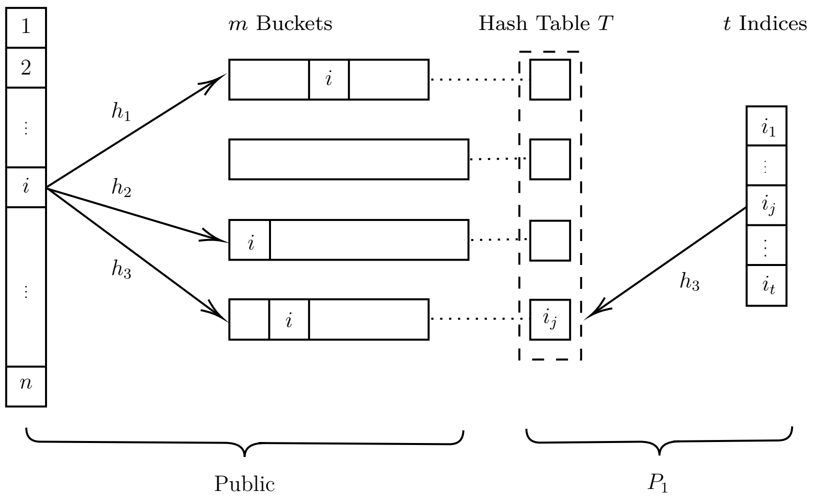


Figure 8: Both parties create m buckets using the same three hash functions. Buckets can have different lengths. The party knowing the MPFSS indices will run cuckoo hashing with the same functions to fill a table T. The placement of an index α_i to a slot j in the table also represent the assignment of an α_i to bucket j.

As a first step for MPFSS using cuckoo hashing w hash functions suitable for cuckoo hashing are selected and the number of buckets is fixed to m. Then, each element from the input domain is hashed using each hash function and inserted into the resulting buckets, similarly to the PSI schemes mentioned in Section 3.2. The size of table T which is used for hashing is set to m and the same w hash functions from before are used. The creation process of buckets ensures, that if an index α_i hashes to the jth slot of table T then α_i is contained in the jth bucket. In the next step, both parties jointly execute m instances of DPF. Each side combines their resulting m vectors to a single MPFSS vector of length n. Figure 8 explains the concept graphically. Amplitude values for the MPFSS can be either secret shared beforehand, known to one party or be randomly chosen.

The MPFSS optimization discussed here requires one party to know all indices. This is a simplification of normal MPFSS discussed earlier. However, for many applications this is sufficient. The simplification is necessary because the party knowing all indices, let it be P_1, runs cuckoo hashing to find an assignment. The hash can be calculated on secret shared values but it is not possible to place an index α_i at position $h(\alpha_i)$ in table T without leaking information. All elements in T could be touched once by the algorithm when a element is inserted or moved. But this also not secure, as the pattern of evictions allows an attacker to draw conclusions.

As discussed in Section 2.5, the thoughtful selection of parameters is important for cuckoo hashing. It would have been possible to realize cuckoo hashing with a stash. But for all elements located on the stash after insertion of the last element a single DPF with the domain n would have been required. The number of element on the stash can also leak information. It would therefore have to be padded to a fixed size which is in $log(t)$. Due to the expected impact on performance as well as the existence of convenient alternatives, the stash is omitted. Consequentially, three hash functions seem to be the right choice. This additionally has the advantage of a high load factor for T. Therefore, the table does not have to be excessively large when trying to accommodate all indices while keeping the failure probability low. This is beneficial as each slot H is directly coupled with one execution of a DPF.

4.2 Secret Sharing of Amplitude Values

Universal MPFSS uses both secret shared indices and amplitude values. As explained earlier, MPFSS based on cuckoo hashing requires indices to be revealed to one party. But using an additional tweak, amplitude values can remain hidden. Each party holds an array $[b]_i$ of length t representing a secret share of the aptitude values. Combining the first element from both sides using XOR will give the amplitude at the position of the lowest index. In other words, XORing the two arrays $[b]_1$ and $[b]_2$ will give all βs in ascending order.
But how does P_2 know, which of the elements from its secret shared array to use for computing the first bucket? P_1 created the assignment to buckets for all indices, therefore it knows their ranks and can choose accordingly. One option to help P_2 are oblivious sorting algorithms [14]. Oblivious indices and bucket numbers are sorted into an ascending order. The pairs are then combined with the corresponding amplitudes. To use this information further it is necessary to sort again by bucket number. All amplitudes and indices remain hidden secret, but the parties will be able to use the correct pairs for each DPF call.

4.3 Asymptotic Runtime Analysis

When talking about MPC protocols it is essential to estimate necessary computation and communication. MPFSS with cuckoo hashing takes $\mathcal{O}(n)$ steps of computation per party.
Communication for m buckets requires:

$$\mathcal{O}\Big(log(m)^2 + m\lambda \cdot log\Big(nk/m + \sqrt{nk \cdot log(m)/m}\Big)\Big)$$

This estimate combines communication necessary for sorting networks, requiring $\mathcal{O}(mlog(m)^2)$ steps [5], and the transfer of seeds between parties. A seed has the length of $\mathcal{O}(\lambda \cdot log(N))$ for a single DPF with the domain size N when using the construction method discussed in [14]. The batching mechanism of MPFSS with cuckoo hashing makes m DPF calls, one for each bucket. Assuming a independent hash functions as discussed in Section 2.5, the bucket creation process can be simplified as placing nk balls into m bin uniformly at random. Following the results from [29], each bucket has at most the length $l = nk/m + \mathcal{O}\Big(\sqrt{nk \cdot log(m)/m}\Big)$ if $n > \mathcal{O}(mlog(m))$. The transmitted data is equivalent to the size of all seeds combined and directly linked to the bucket lengths. Here, all buckets are assumed to have maximal possible length. Therefore, the total amount of data communicated is in $\mathcal{O}\Big(m\lambda \cdot log(l)\Big)$. When simplifying the terms it becomes clear, that communication requirements for both algorithms is very similar. For computation on the other hand MPFSS Cuckoo performs better asymptotically.

4.4 Security

This section attempts to give a general understanding of how security in the protocol works. As attacker model the semi-honest case is used.

MPFSS builds on single point FSS for DPF as discussed in Section 2.4.1. For each bucket, a single DPF is executed. DPF is used as a black box, its security therefore directly follows from the proof of Doerner et al. in [14].

So it remains to prove, that the cuckoo hashing step does not leak information. This only happens when a hashing failure occurs and $MaxLoop$ is reached. Knowing hashing was not successful for a certain set of buckets allows an attacker to find possible sets of indices trough searching for circles in the graph of evictions. The failure probability therefore has to be kept as small as possible. It can be influenced by adjusting cuckoo hashing parameters. This work establishes a failure probability of 2^{-40} which is negligible.

4.5 Applications

4.5.1 Floram

Oblivious random access memory (ORAM)[18] is a construction for data-dependent access memory where a server does not gain information about what data a client accesses. Read and write operations are masked in a way that monitoring physical memory becomes infeasible and any two accesses from the same client are indistinguishable. Although ORAM might seem similar to the earlier mentioned PIR, it differs as ORAM additionally provides writing functionality and attempts to hide not only accesses but also the content of stored data from the server[14].

Doerner et al. proposed *function-secret-sharing linear ORAM* (Floram)[14], an oblivious memory based on FSS. It is a type of distributed ORAM, meaning its database DB is located on m independent servers.

The approach requires two different memory representations for reading and writing. The authors use additional data structures to periodically transfer changes. For writing, the memory is spread over two servers S_1, S_2 each holding a secret share of DB. Write operations use a DPF which will evaluate to the XOR-difference between new and old value at the target location i. Each server will receive one of the keys that were created by the *Gen* algorithm. The memory's size is the input domain of the DPF. Servers will expand their key and XOR the resulting vector with their memory share.
For reading, each server hold a copy of DB masked by a pseudorandom key, which is only known to the client. Again, the client generates a set of keys using i, the index of the location to be read. Servers evaluate their key, receiving a secret share of the DPF vector and the so-called advice bits, a binary secret shared vector of the same point function. Both servers then XOR their memory with the advice bit vector and return the result. XORing both results will give the client a masked value of location i. But since it knows the key with which the memory was obscured, it can recover the actual value.

Using MPFSS instead of DPF here allows more efficient memory accesses multiple sequential access are planned. Either a series of read or a sequence of write operations can be compressed this way. So for the case where a Floram client know the indices MPFSS using cuckoo hashing can improve performance.

4.5.2 VOLE

Oblivious linear-function evaluation (OLE) allows one party P_1 to learn, for an arbitrary element x, the linear combination $w = u \cdot x + v$ of two elements u, v held by P_2. When u and v are not simply numbers, but vectors of the same size, the method is called *vector oblivious linear-function evaluation* (VOLE) [4]. For this purpose, secret correlated randomness plays an essential part. This can be realized trough *multiplication triples.* Bolye et al. [9] proposed the use of FSS for compressing correlated random vectors which are used for the multiplication triplets. More precisely, they focused on *VOLE correlation.* In this case, neither party holds any vectors or parameters at the beginning. Instead party P_1 receives $\vec{u}$ and $\vec{v}$ as output while P_2 gets x and $\vec{w}$.

The MPFSS scheme necessary for the protocol is less strict than approach discussed in this thesis, as it does not require indices or amplitude values to be secret shared. The MPFSS protocol in the focus of this thesis returns XOR secret shares, while VOLE requires secret shares to be additive. However, it is possible to convert representations of such shares into one another [31, Appenix].
An main advantage of using MPFSS for VOLE is the smaller seed size resulting in reduced interaction between the parties. Vector operations are an essential part of machine learning. So providing efficient, oblivious vector operations is a step towards secure machine learning. Inputs remain secret but useful models can be created. Additionally, one single run of VOLE can replace a large number of OLE executions. This is useful for example in arithmetic circuit evaluation for which relies on OLE [9]. Another application is *oblivious polynomial evaluation* (OPE). This technique can be implemented using VOLE [23] and has many applications for example in privacy preserving set operations and data compression.

5 Implementation

5.1 Obliv-C

Obliv-C [35, 24] is an extension of the C language which simplifies the implementation of MPC programs. It provides abstractions and allows programmers to write code and create protocols without having to define garbled circuits themselves. It compiles C code to Yao garbled circuits.
Obliv-C introduces special types for data shared between parties in a way that its actual values can only be retrieved trough collaboration. This data is called *oblivious* or short *obliv*. Such new types are for example **obliv int** and **obliv char**. All of these data types build on the notion of oblivious bits.
Control structures, such as **obliv if**, are also provided. This way **obliv** variables can be used in conditional statements.
Using Obliv-C has many advantages. First, is greatly simplifies the process of secret sharing information. Since it is reasonably widespread, libraries for Obliv-C exist which implement advanced algorithms. For example the library "Absentminded Crypto Kit" (ACKLib)[14] provides many additional functionalities. Among other things, it implements distributed point functions and Batcher merge-sort for sorting networks.
There are many pitfalls when using the C language and its extensions, which can make debugging tiresome. Therefore, Obliv-C code was called from a C++ environment. Creation of buckets and the assignment of indices was realized in C++, while oblivious sorting and calls to the DPF had to be realized in Obliv-C. Results are combined and prepared for further processing in C++. Bazel was used as build system and to create docker images.

5.2 Hashing

Performance of the cuckoo hashing algorithm greatly depends on the choice of hash functions. It is important to differentiate between a hashing algorithm (like cuckoo hashing) and a hash function (e.g. the function used by cuckoo hashing to place elements).
Following the results of Fotakis et al. [15] it is evident to select truly random hash functions instead of choosing from a $(log(1), log(n))$-universal hash family. One option are cryptographic functions like `SHA256`. But these are known to be time and computation intensive. To simplify hashing and to ensure good performance, the Abseil framework[1] was used. The associated hashing functionality provides efficient hashing with somewhat random output. It is important to note, that the Abseil documentation mentions it it not guaranteed both parties will use the same hash function when calling `abseil::hash()`. So there is a risk that different hash functions result in different buckets and break the algorithm. So far, this has not been an issue even after tests on various machines. There are many implementations of `hash()` in C available, for example Boost [6]. Therefore swapping the library at any later point will be straightforward.

Algorithm 1 Pseudo-code for cuckoo hashing as used by MPFSS Cuckoo. The symbol $\|$ represents the concatenation of two elements.

Parameters P_1**:** m, t, $\mathbf{a}$, $\mathbf{r}$
Setup:

- $w \leftarrow 3$
- $MaxLoop \leftarrow 0.5t$
- $\mathbf{r}$ is vector of w random values, initialized beforehand
- Initialize hash table T with length m
- Initialize table of Boolean values B with length m

```
 1: function Place($v$, $\mathbf{r}$, $cnt$)
 2:     $j \leftarrow$ Rand() mod $w$
 3:     $r_j \leftarrow \mathbf{r}[j]$
 4:     $pos \leftarrow$ Hash($v \| r_j$)
 5:     if $B[pos] = 0$ then
 6:         $T[pos] \leftarrow v$
 7:         $B[pos] \leftarrow 1$
 8:     else
 9:         $cnt \leftarrow cnt + 1$
10:         if $cnt = MaxLoop$ then
11:             Error State
12:         else
13:             $v_{new} \leftarrow T[pos]$
14:             $T[pos] \leftarrow v$
15:             Place($v_{new}$, $\mathbf{r}$, $cnt$)
```

CuckooHashing(a, t**, r)**

```
16: Initialize random number generator Rand()
17: for $i \in [0, t-1]$ do
18:     $cnt \leftarrow 0$
19:     Place($\mathbf{a}[i]$, $\mathbf{r}$, $cnt$)
20: return $T$
```

To ensure independence between the w hash functions, w random values are picked. Keys are always hashed in a pair with the random value associated to the selected hash function. Algorithm 1 shows pseudo-code for cuckoo hashing as used by MPFSS Cuckoo.

5.3 MPFSS Cuckoo

The MPFSS Cuckoo algorithm takes a list of indices and a secret share of amplitude values as input for party P_1. For P_2 only the amplitude secret share needs to be provided. Then, a connection between the two parties is established. The following steps are also described as pseudo-code in Algorithm 2. Both sides use the same seed to create a random value for each hash function, to ensure independence between the hash function. Because hash functions are now fixed, the parties separately start creating buckets using simple hashing. Since both use the same functions, they will obtain the same results. In the next step, P_1 tries to find an assignment for its indices. It employs cuckoo hashing using the same three hash functions as used for simple hashing. Now, the Obliv-C protocol can start. First indices are combined with the corresponding amplitude values as discussed in Section 4.2. Next, one DPF per bucket is called. This step was parallelized, so that multiple threads each processed a subset of DPF calls. The ACKLib provides two version of DPF using different optimizations. The `fss_cprg` variant is significantly more efficient for domain sizes smaller than 2^{25}. For a domain size of 2^{30}, the advantage over `fss` becomes clear [14]. Because the evaluation only deals with $n < 2^25$ `fss_cprg` is used to realize MPFSS.

Doerner and Shelat implemented for [14] and the ACKLIB a DPF schema using Obliv-C following the technique discussed in Section 2.4.1. This implementation combines *Gen* and *Eval* into one function. It requires the value of index α and the domain size n as input. To each party the function returns a vector v_{bool} of length n containing the advice bits of the DPF. The parties will also receive v_{val} of the same length consisting of 128 bit integers. Both vectors represent secret shares. The input function can be reconstructed by XORing both v_{val} vectors. So $v^x_{val,1} \oplus v^x_{val,2} = f(x)$ for $x \in [0, n-1]$. The vector reconstructed from the two v_{bool} will be 1 where $f(x) \neq 0$ and can be used to verify correctness of the DPF. If for example the amplitude value β is zero, v_{bool} will indicate which index was selected while v_{val} will be a vector of zeros. Unlike the theoretical construction, the implementation of Doerner and Shelat does not allow specification of the amplitude value β. Instead, β will be set randomly during execution and returned as 128 bit value. The value therefore has to be corrected. For each $i \in [0, n]$ both parties recalculate v_{val} using the following equation:

$$v^i_{val} = v^i_{val} \oplus (v^i_{bool} \cdot (\beta_{random} \oplus \beta))$$

When all DPFs successfully executed, the respective resulting vectors $v^1_{val}, \cdots, v^m_{val}$ are combined to a single MPFSS with length n. To facilitate black box usage, the program is callable using a C++ API. To allow amortization of bucket setup costs, buckets created by the MPFSS are returned to the caller. These can be provided again for the next execution with the same domain size n.

Algorithm 2 This algorithm describes the steps necessary for MPFSS using cuckoo hashing. Oblivious variables are indicated trough having *obliv* as part of their name and being written in red. Functions $e_1()$ and $e_2()$ represent the calculation of e with different parameters.

Parameters P_1: n, t, $\mathbf{a}$, $\mathbf{b}_1$
Parameters P_2: n, t, $\mathbf{b}_2$
Setup:

- $w \leftarrow 3$
- $\lambda \leftarrow 40$ ▷ Security parameter
- **if** $t > 512$ **then**
  $m \leftarrow e_1(t) \cdot t$
  else if t>4
  $m \leftarrow e_2(t) \cdot t$
  else
  $m \leftarrow e_2(t = 4)$
- Both parties use the same seed to initialize $\mathbf{r}$ with w random values
- The parties have established a connection for communication

MPFSSCuckoo(n, t):

```
1:  function CreateBuckets
2:      Initialize $m$ empty Buckets $M_1, \cdots, M_m$
3:      for $i \in [0, n-1]$ do
4:          for $j \in [0, w-1]$ do
5:              $r_j \leftarrow \mathbf{r}[j]$
6:              $h_{i,j} \leftarrow \text{Hash}(i \parallel r_j)$
7:              Insert $i$ into bucket $M_{h_{i,j}}$
8:      return $M_1, \cdots, M_m$

9:  function CreateAssignment
10:     if Party $P_1$ then
11:         $T \leftarrow \text{CuckooHashing}(\mathbf{a}, t, \mathbf{r})$
12:     return $T$
```

```
function AssignBetaShares($T$)
    $obliv\_B \leftarrow \mathbf{b}_1 \oplus \mathbf{b}_2$                ▷ Creating oblivious values held by both parties
    $obliv\_A \leftarrow T$
    for $i \in [0, m-1]$ do
        $bucket\_no \leftarrow i$
        $obliv\_\alpha_i \leftarrow obliv\_A$ at position $i$
        Make pairs $[obliv\_\alpha_i, bucket\_no]$
    Sort pairs by $obliv\_\alpha$ in ascending order
    for $i \in [0, m-1]$ do
        $obliv\_\beta_i \leftarrow obliv\_B$ at position $i$
        Make tuples by adding $obliv\_\beta_i$ to the $i$-th pair
    Sort tuples by $bucket\_no$ in ascending order
    return tuples as $matches$

function main
    $M_1, \cdots, M_m \leftarrow$ CreateBuckets()
    $T \leftarrow$ CreateAssignment()
                                                          ▷ Start of Obliv-C part
    $matches \leftarrow$ AssignBetaShares($T$)
    for $i \in [0, m-1]$ do                          ▷ This loop can be multi-threaded
        $bucket\_size_i \leftarrow$ Size of $M_i$
        $obliv\_\alpha_i$, $obliv\_\beta_i \leftarrow matches_i$
        $\vec{v_i} \leftarrow DPF(obliv\_\alpha_i, obliv\_\beta_i, bucket\_size_i)$
    Initialize empty vector $mpfss$ of length $n$
    for $i \in [0, m-1]$ do
        $bucket\_size_i \leftarrow$ Size of $M_i$
        for $j \in [0, bucket\_size_i - 1]$ do
            $val \leftarrow M_{i,j}$
            $mpfss_{val} \leftarrow mpfss_{val} \oplus v_{i,j}$
    return $mpfss$
```

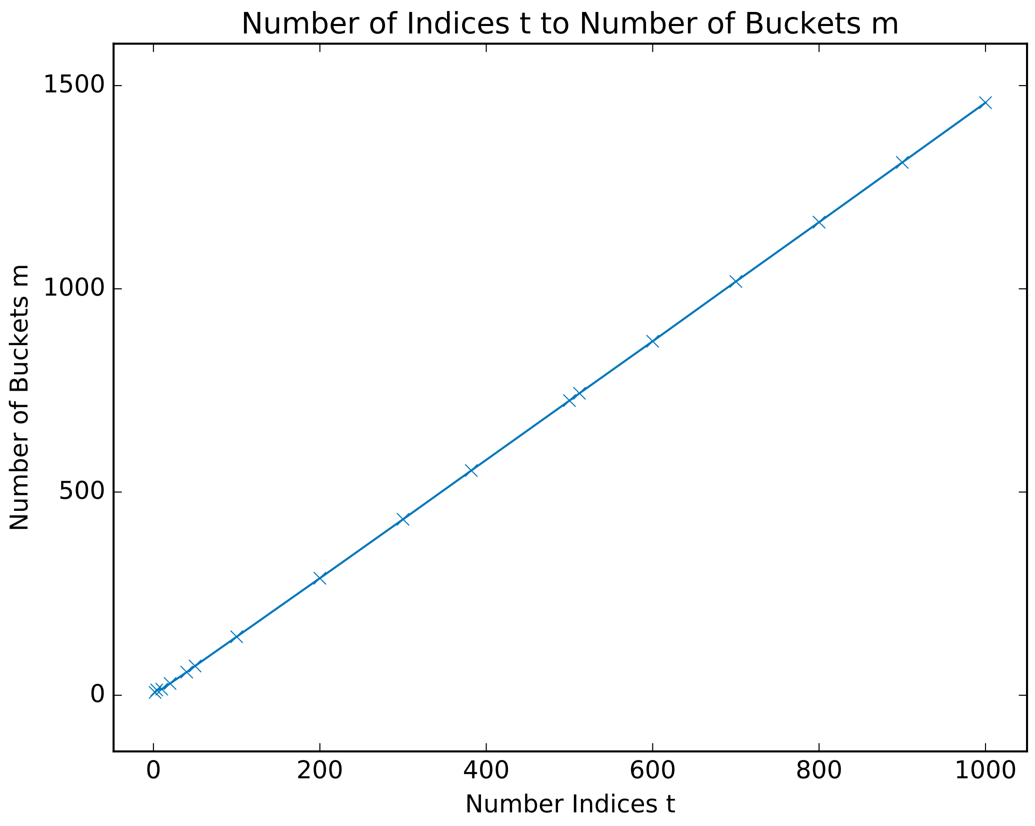


Figure 9: This plot shows the corresponding number of buckets m for different values of t.

5.4 Parameter Choice

The number of buckets m is essential for keeping failure probability of cuckoo hashing low. In the appendix of [3] Angel et al. provide an empiric functions for calculating the necessary size of a cuckoo hash table dependent on the number of elements inserted:

$$e = \lambda/a_n - b_n/a_n \tag{1}$$

Parameter e represents the cuckoo table expansion and $m = e \cdot t$. For the case $t \leq 512$ the constants are:

$$a_n = 123.5$$
$$b_n = -130 - log_2(t)$$

If $4 < t < 512$, the formula becomes more difficult. Then

$$a_n = 123.5 \cdot CDF_{normal}(x = t, \mu = 6.3, \sigma = 2.3)$$
$$b_n = -130 \cdot CDF_{normal}(x = t, \mu = 6.45, \sigma = 2.18) - log_2(t)$$

CDF_{normal} is the cumulative distribution function over a normal distribution with the given mean μ and standard deviation σ. The uncovered case of $t < 4$ was considered unlikely and not target of optimization. Therefore, the same value as if $t = 4$ is used. Figure 9 visualizes this relationship between t and m.

5.5 Multi-Threading

There are multiple options on how to speed up the MPFSS Cuckoo algorithm by using multi-threading. A good starting point is to parallelize the DPF executions over all buckets after indices were sorted (see Section 4.2). An advantage of this method is the a single call to `execYaoProtocol()`. This function starts all Obliv-C protocols and produces a small overhead.
OpenMP[25] is a widely used API facilitating multi-threading in C and C++ programs. Obliv-C itself is not compatible with OpenMP, but this problem can be solved by enabling multi-threading only in files compiled with a standard C compiler. `structs` are used as function pointer to call Obliv-C functions from the multi-threaded code. This workaround was not directly applicable to the implementation. It was necessary to fork the ACKLib and rewrite some parts of the library. All calls to `ocBroadcast` in the relevant functions had to be replaced with one call to `feedObliv` and one to `revealObliv` each. This is not as efficient as the original version, but ensures synchronization between both parties. Direct calls to `pthreads` were tested, but resulted in the same problems as with OpenMP.
Running parallelization in C++ (so before executing the Obliv-C Yao protocol) was considered but ruled out. Since sorting networks for finding pairs of indices and amplitude values is realized in Obliv-C, all threads would independently have to execute this step causing a significant, measurable overhead.
Each thread requires a separate TCP connection. Since splitting a existing connection of a running Obliv-C protocol takes up to 60 ms, the number of splits was kept as low as possible. Therefore, there are only as many threads as there are available cores. Each tread sequentially runs fraction of the DPF calls.

5.6 MPFSS Naive

For comparison, the naive approach to MPFSS was also implemented. The corresponding pseudo-code is shown in Algorithm 3. In this version, Party P_1 knows the indices. The algorithm allows stronger security guarantees but since it was implemented for comparison it was given the same functionality as MPFSS Cuckoo. Both parties hold their respective share of amplitude values. After establishing a connection, the Obliv-C protocol is directly called. To ensure that indices are matched with the correct amplitude values, P_1 sorts $\mathbf{a}$. Then both sides combine their amplitude shares to receive the actual values. These are of the data type *obliv*, so neither party learns the actual value. In the net step one DPF is called for each index. Results have to be corrected as described in Section 5.3. Finally, each party combines their t DPF vector to one MPFSS vector of size n.

Algorithm 3 This algorithm explains a simple version of MPFSS, also called MPFSS Naive. Oblivious variables are indicated trough having *obliv* as part of their name and being written in red.

```
Parameters $P_1$: $n$, $t$, $\mathbf{a}$, $\mathbf{b}_1$
Parameters $P_2$: $n$, $t$, $\mathbf{b}_2$
MPFSSNaive($n, t$):
 1: if Party $P_1$ then
 2:     Sort $\mathbf{a}$ in ascending order
 3: $obliv\_A \leftarrow \mathbf{a}$
 4: $obliv\_B \leftarrow \mathbf{b}_1 \oplus \mathbf{b}_2$
 5: for $i \in [0, t-1]$ do                         ▷ This loop can be multi-threaded
 6:     $obliv\_\alpha_i \leftarrow obliv\_A$ at position $i$
 7:     $obliv\_\beta_i \leftarrow obliv\_B$ at position $i$
 8:     $\vec{v_i} \leftarrow DPF(obliv\_\alpha_i, obliv\_\beta_i, n)$
 9: Initialize vector $mpfss$ of length $n$
10: for $i \in [0, t-1]$ do
11:     for $j \in [0, n-1]$ do
12:         $mpfss_{i,j} \leftarrow mpfss_{i,j} \oplus v_{i,j}$
13: return $mpfss$
```

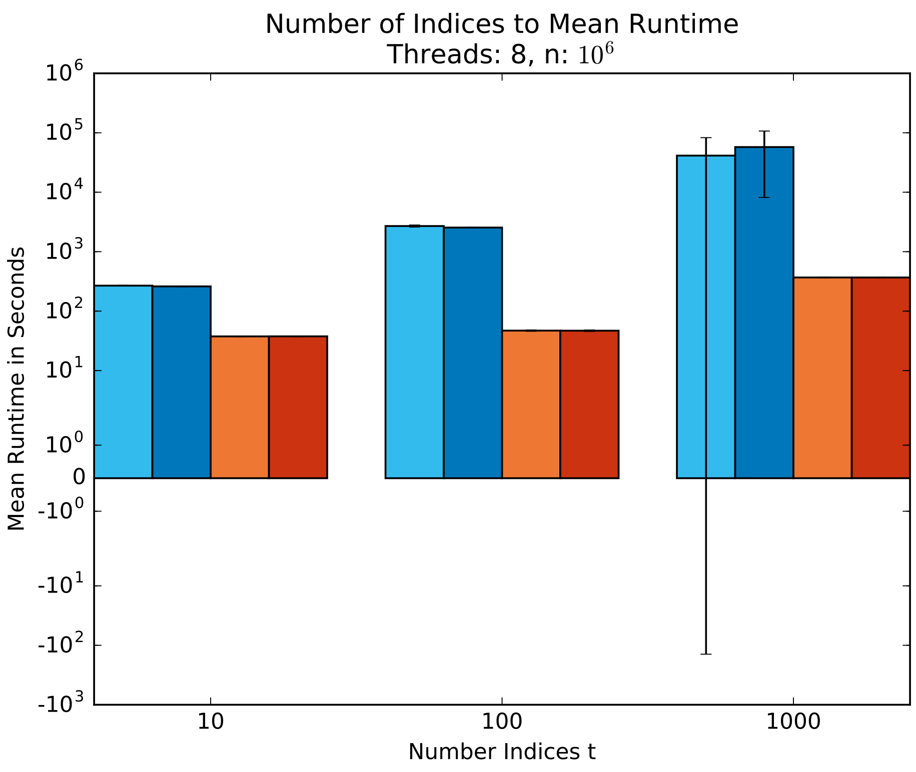
6 Experimental Evaluation

In this section, the performance of MPFSS with cuckoo hashing is evaluated and compared to the naive variant. Experiments were conducted on virtual machines (VMs) created through Microsoft's cloud provider Azure. More specifically, "Standard_B8ms" VMs were selected. These are general purpose machines with 8 cores, 32 GB RAM and an Intel Xeon CPU (E5-2673 v3 @ 2.40GHz)[12]. Machines, that communicated with one another over LAN, were placed in the same "Azure-Region" (generally North Europe) and located in the same subnet. This means, they are not necessarily placed in the same data center. Therefore, ping measurements were conducted to verify connection quality. The mean round trip time (RTT) between two machines was 2.1 ms with a standard deviation (std) of 3.3 ms. Docker images were distributed to the VMs using a GitLab registry. All images were compiled using the compiler optimization.

6.1 Runtime Measurements

Two implementations were compared: MPFSS Naive, shown in Algorithm 3 and MPFSS Cuckoo, explained in Algorithm 2. Multi-threading was enabled for the relevant parts of each program. The number of parallel threads was set to 8, unless specified otherwise. Runtime measurements started right before the transition to Obliv-C and stopped, when the Obliv-C protocol exited. For MPFSS Cuckoo, this means that bucket creation and finding an assignment were not included. These aspects had little influence on the runtime and are discussed separately later. Generating the magnitude values from secret shares as mentioned in Section 4.2 on the other hand is realized in Obliv-C and therefore part of the runtime measurement. For both algorithms, the creation of a final MPFSS vector contributed to the measurement.

A protocol instance always consists of two parties communicating. Therefore, when measuring the runtime for a single instance there are two results, one for each party. When two parties run a single instance of DPF a difference in runtime can be seen. Since party P_1 creates the circuit, it finishes faster. This had impact on measurements of MPFSS Naive (see Figure 10). Because of higher variance this becomes less important for larger values of n and t. Due to shorter domain sizes of single DPFs and additional synchronization, the effect can not be seen in MPFSS Cuckoo. Graphs of MPFSS Cuckoo in this chapter therefore include both measurements each run. For MPFSS Naive the results of P_2 are used. Error bars in plots represent the standard deviation.



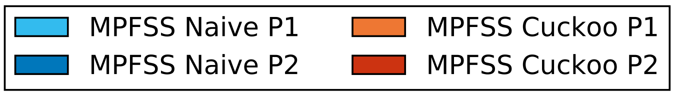


Figure 10: This figure compares runtime differences between P_1 and P_2 for both protocols using various t and $n = 10^6$. The first column for $t = 1,000$ represents a $mean = 41,304$ with $std = 41,445$, while the second column in the same group has $mean = 57,504$ and $std = 49,313$. The logarithmic y-scale changes the proportions of error bars. Measurements of the for $t < 1,000$ consist of at least 14 measurements. For $t = 1,000$, one bar represents 6-7 measurements.

When multiple, sequential runs of MPFSS Cuckoo had the same parameters, buckets were reused. Since measurements with $t > 1,000$ can take hours or days to finish, not all measurements could be repeated the same amount of times for both implementations. For this reason, in some cases numbers next to data points indicate how often a measurement was repeated.

To determine general differences in performance, a reasonably large value for n ($n = 10^6$) was selected and both algorithms were run using a range of t values. The goal was to determine, for which $r = t/n$ MPFSS Cuckoo is faster than MPFSS Naive. This is especially interesting when using MPFSS in the context of combining multiple DPF calls into one MPFSS instance e.g. for reading or writing to a database (see Section

4.5). Figure 11 shows the results on a double-logarithmic scale plot. It clearly shows, that MPFSS Cuckoo performs better than the alternative for all measured ratios. At about 0.01% the graph of MPFSS Cuckoo bends. Before the bend, runtime grows slowly. Afterward, it increases with the same speed as the naive approach. The MPFSS Naive graph rises in a monotonous manner and forms an almost straight line. When plotting the measurements on conventional scale, it becomes clear that the runtime of both algorithms grow linearly for large n. This plot, as well as an enlargement of results for MPFSS Cuckoo with $t < 500$ can be found in the appendix (see Section 9). Most measurements for MPFSS Naive with $20 \leq t \leq 90$ were part of one series conducted during a single night. Offset and variance for this part of the graph indicate high load on the network or management processes initiated by Azure and are most likely not caused by the implementation.

To find exact values for which the batching overhead becomes too large to be faster than the naive algorithm, lower values of n and t were tested. Figure 12 compares the runtime of both algorithms using $10^4 < n < 3 \cdot 10^4$ and $t < 20$. For lower n, runtime of MPFSS Naive increases more slowly. The results for MPFSS Cuckoo indicate little dependence on n.

The scaling measurements from above were repeated over the Internet. One machine was placed in North Europe, another on the east-side of the United States. Measured round-trip time between these machines was 73 ms with a standard deviation of 12.6 ms. A problem when conducting measurements over the Internet was that the TCP connections were instable. Therefore, additional error handling was necessary. Results are shown in Figure 13. Since load on the Internet is volatile, these measurements have a higher standard deviation. The increased RTT causes the intersection of graphs to be shifted towards larger values of t. For $n > 3 \cdot 10^4$ the improvement compared to the naive approach shows more clearly. For $n = 10^6$ there no intersection even for small values of t. Increasing n seems to shift the line for MPFSS Cuckoo in the y-dimension. For MPFSS Naive the slope changes and becomes more steep. The plot also visualizes setup costs for MPFSS Cuckoo, as its runtime does not start rising until $t = 10$. These costs consist of one FSS tree per bucket that has to be generated and sent. It also shows, how choice of n and t influence the intersection.

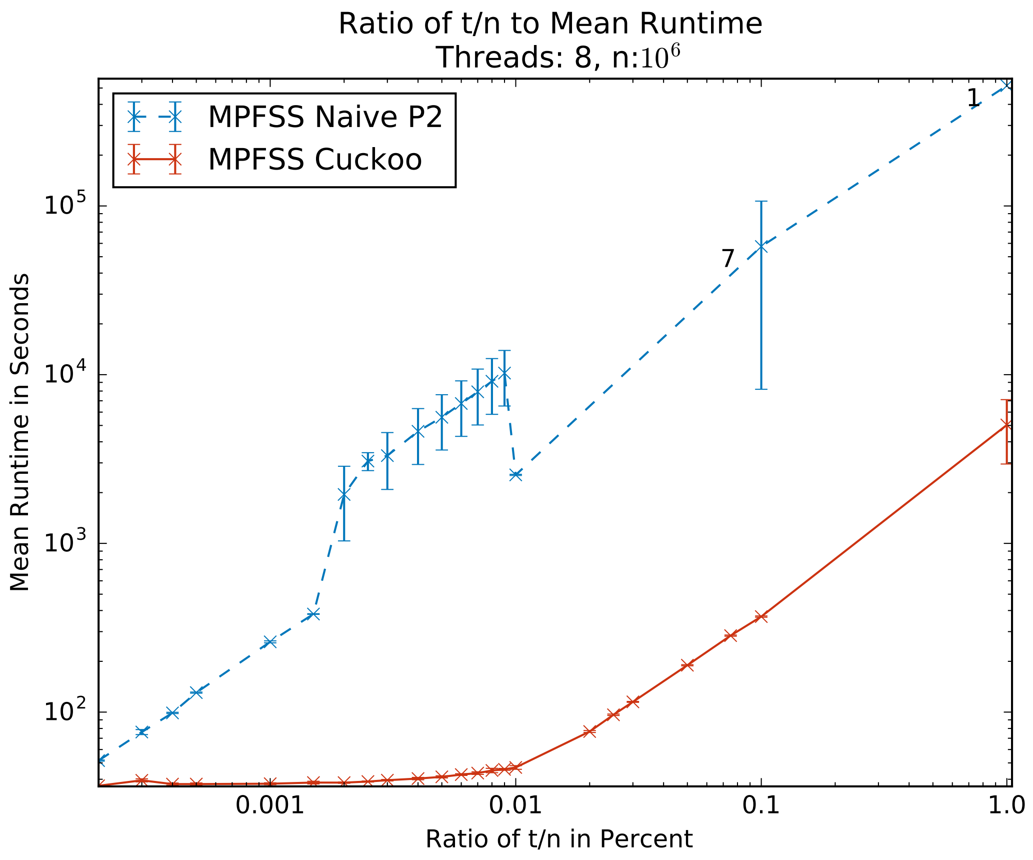


Figure 11: Runtime measurement of MPFSS Cuckoo and MPFSS Naive. The domain size was fixed to $n = 10^6$ while multiple values for t were tested. All data points, unless annotated otherwise, consist of at least 10 measurements. Error bars represent the standard deviation. Most measurements for MPFSS Naive with $20 \leq t \leq 90$ were part of one measurement series conducted during a single night and seem to have been heavily influenced by outside factors.

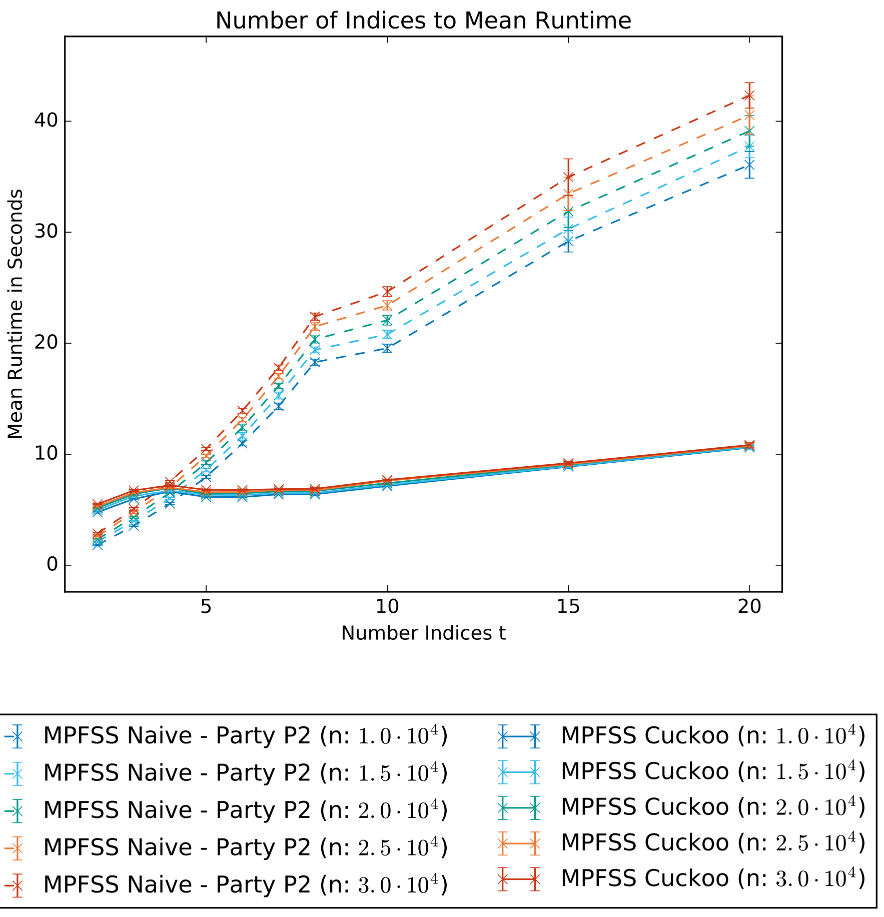


Figure 12: Runtime measurements of MPFSS Cuckoo and MPFSS Naive for low values of n and t. Each data point consists of at least 14 measurements. Error bars represent the standard deviation.

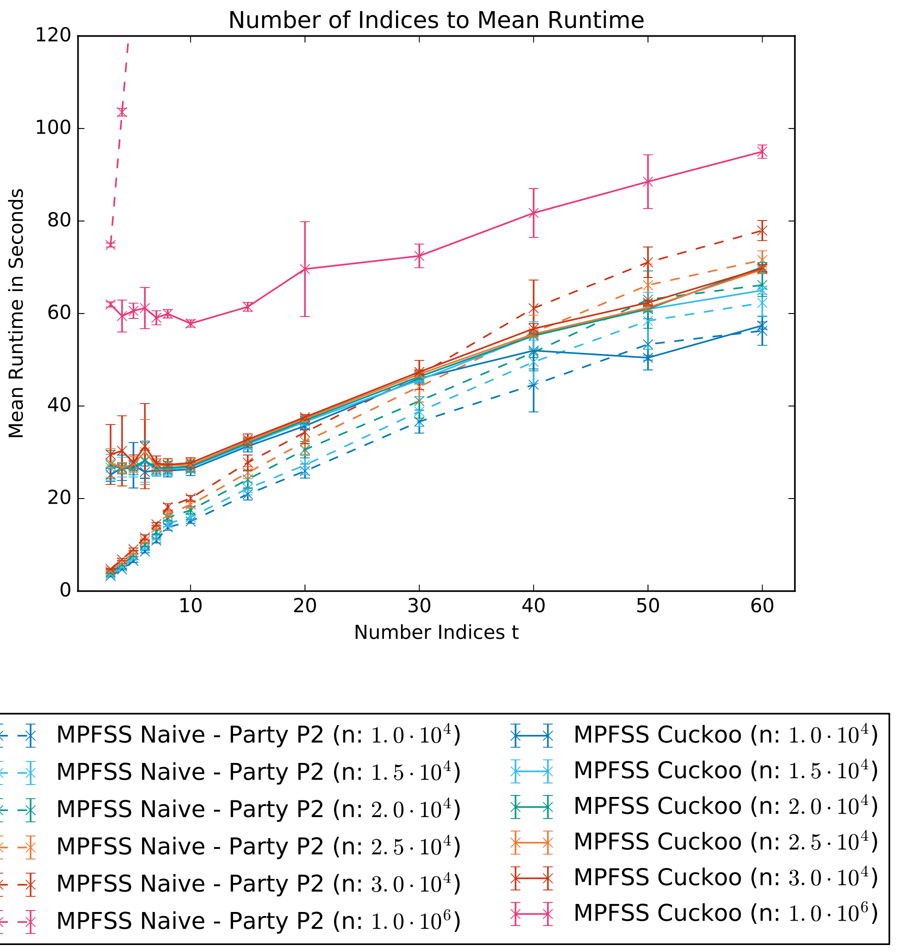


Figure 13: Runtime measurements of MPFSS Cuckoo and MPFSS Naive over the Internet for a low number of indices t and various domain sizes n. Each data point consists of at least 10 measurements. Error bars represent the standard deviation.

Another experiment evaluated the runtime using values for n and t from [9]. The exact values of both parameters are shown in Table 1. These are parameters for Boyle et al.'s VOLE G_{primal} generator and were optimized for security. Figure 14 shows, that the cuckoo hashing based version performs significantly better. Its runtime increases slower than for the naive algorithm.

n	2^{11}	2^{14}	2^{16}	2^{18}	2^{20}	2^{24}
t	74	192	382	741	1,422	5,202

Table 1: Parameters for VOLE G_{primal} generator of Boyle et al. which are optimized for security[9].

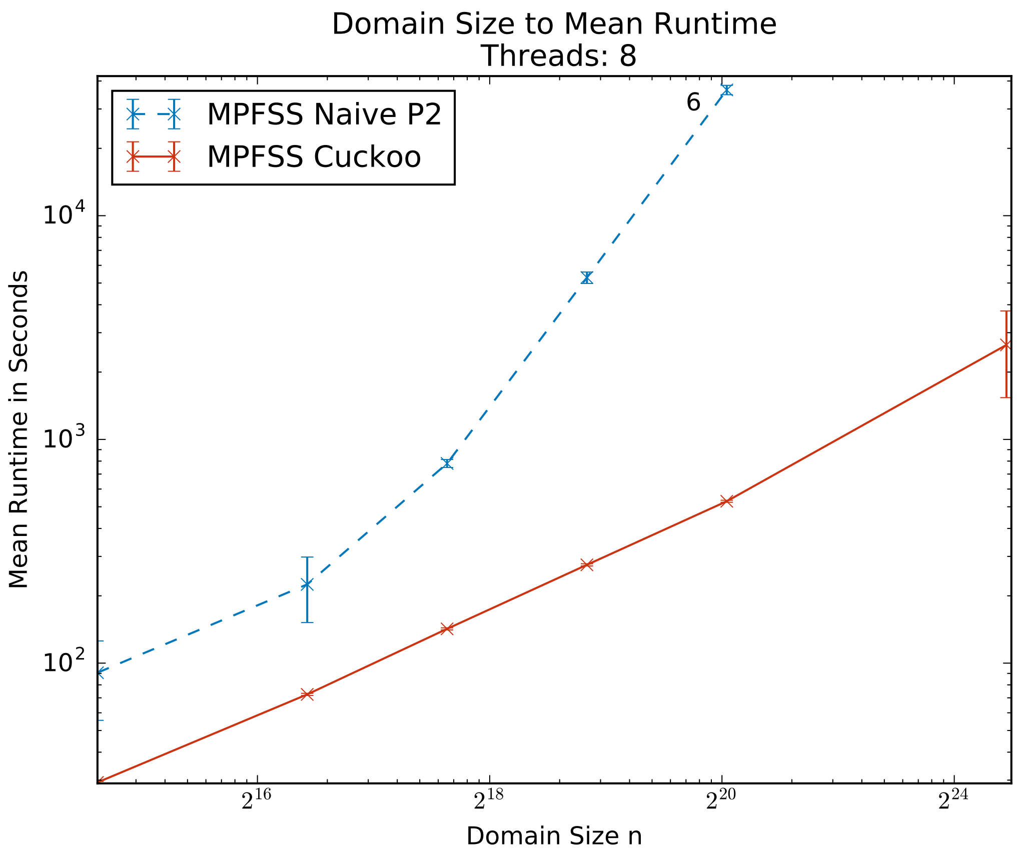


Figure 14: Runtime measurements of MPFSS Cuckoo and MPFSS Naive for VOLE parameters from Table 1 plotted on double-logarithmic scale. Unless annotated otherwise, each data point consists of 12 measurements. Error bars represent the standard deviation.

Since multi-threading is enabled for MPFSS Cuckoo, quantifying its impact is inevitable. Figure 15 visualizes the runtime for different parameters of t with a fixed n

and different numbers of threads. For larger t, the impact is more visible. The curve flattens fast, making it difficult to identify improvement when comparing results for 4 and 8 threads.

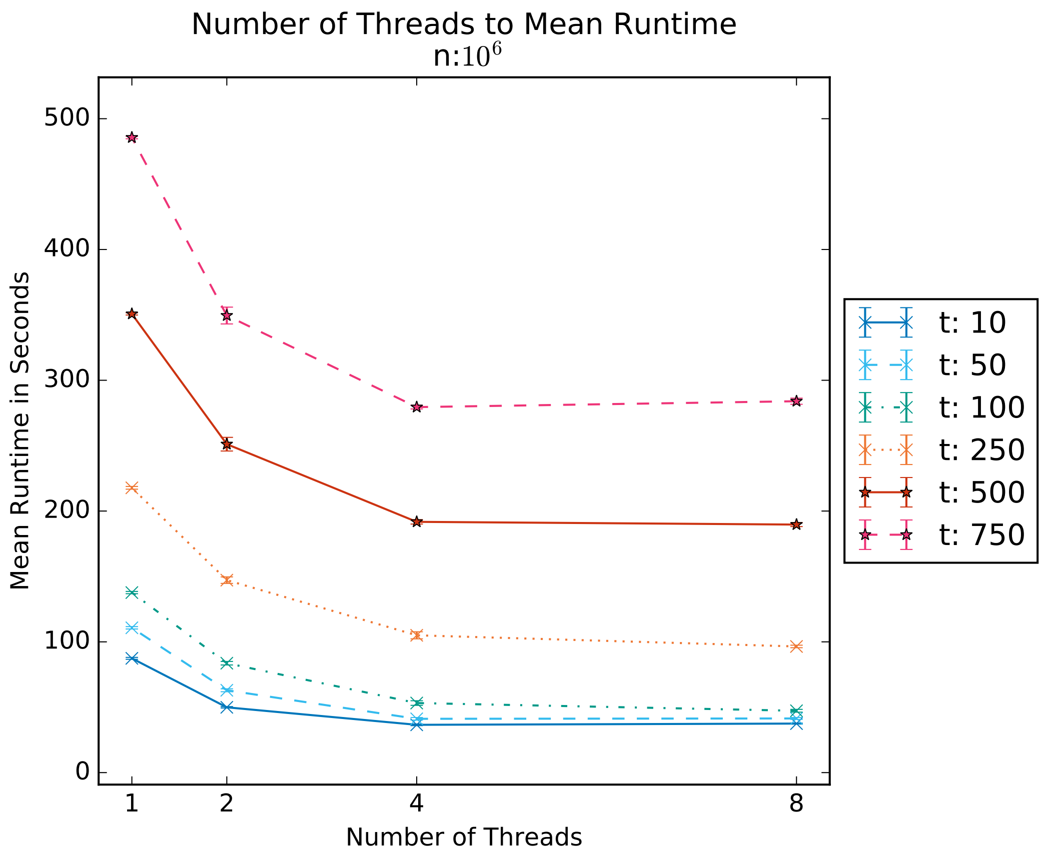


Figure 15: Runtime measurements of MPFSS Cuckoo for various t and fixed n with different numbers of available threads. Each data point consists of 26 to 30 runs. Error bars represent the standard deviation.

6.2 Setup Costs and Sorting Networks

Preparation time for MPFSS Cuckoo were measured separately. The time used for Bucket creation depends on the value of t. Even for a large ratio of $r = 0.01$ (so 1% of the domain size are selected as indices) and $n = 10^6$, this step took less than 150 ms. For the same parameters, an assignment was found within less than 15 ms.

Creating pairs of indices and amplitude values for MPFSS Cuckoo is included in the measurement of Section 6.1. But since it also depends on t and n, additional separate experiments were conducted. Table 2 shows the results of these measurements.

t	Mean	Std	Mean Runtime Protocol
30	0.0561	0.005	39.5
300	1.26	0.11	115.0
500	2.49	0.18	189.6
1,000	6.18	0.48	368.3

Table 2: Runtime measurements of sorting network for $n = 10^6$ and variable t. All values are given in seconds. "Mean" refers to the runtime required for sorting, while "Mean Runtime Protocol" attempts to give an idea how long the whole execution of MPFSS Cuckoo with the same parameters takes.

7 Discussion

The experimental evaluation has shown that MPFSS using cuckoo hashing performs better than the naive variant. MPFSS Naive is faster only for low values of n and t. This is due to the large DPF setup costs of MPFSS Cuckoo as it initiates a greater number of DPF calls. More specifically, it makes one DPF call per bucket, so in total m instead of t. On the other hand, these calls use a significantly shorter domains size, which compensate for tree setup costs. MPFSS Cuckoo shows good results for high ratios of t/n. This is relevant for applications where multiple DPFs can be combined into one multi-point function for performance gains. The evaluations have shown that multi-threading for MPFSS Cuckoo is useful. But the number of threads has to be selected carefully. For example for $t \leq 750$, there is no advantage of using more than 4 threads. This is most likely due to the overhead created by thread management and the theoretical bound for speed-up trough threading given by Amdahl's Law [2].

Runtime measurements were conducted in a LAN and over the Internet. Runs over the Internet for either protocol failed more often due to broken TCP connections. The longer RTT causes setup costs to have a stronger influence than in a LAN. This causes the algorithm to perform worse than the naive approach for small parameters where DPF setup costs are not balanced out by the shorter domain sizes.

For finding the correct pairs of indices and amplitudes, the implementation employs sorting networks. A possible improvement would be to use *permutation networks* instead. If one party knows the new order of elements, oblivious permutation networks allow to rearrange values of the other party without revealing the permutation. Since P_1 knows all indices, it can sort them in ascending order and get the required permutation. Using sorting networks takes $\mathcal{O}(nlog(n)^2)$, while Waksman permutation networks have a runtime of $\mathcal{O}(n \cdot log(n))$[20]. But as shown in the experimental evaluation in Table 2, the sorting step has little influence on the allover runtime of MPFSS Cuckoo.

One requirement for MPFSS Cuckoo was, that one party knows the true value of all indices. It would be possible to realize the cuckoo hashing process using an ORAM, which allows write operations to oblivious positions. Implementations for ORAM in Obliv-C exist. Then, indices could remain secret shared. Further research to determine the applicability of this approach and its related cost would be necessary.

8 Conclusion

This work compared a batching based multi-point function secret sharing (MPFSS) algorithm using cuckoo hashing to a naive approach. The goal was to quantify the performance gain achieved through batching and use of cuckoo hashing. Asymptotically, MPFSS Cuckoo is faster during local computation. Experiments have shown that MPFSS Cuckoo is significantly faster than a naive approach for large n both in the LAN and over the Internet. The exact point at which the usage of MPFSS Cuckoo pays off depends on t, n and the round trip time between both parties. Compared to the naive approach, it has the disadvantage of being less general. To perform MPFSS

Cuckoo, it is essential for one party to know the indices at which the multi-point function is not equal zero.

Despite this constraint and because of its better performance for the relevant parameters, the batching based approach is suitable for black box use to speed up VOLE as source of correlated randomness.
The optimized algorithm for MPFSS with t indices performs better than the equivalent amount of DPF runs. It can therefore be used to improve performance for applications with multiple sequential runs of DPF. For example Floram can be enhanced by combining a set of DPFs into a single instance of MPFSS. MPFSS Cuckoo performs significantly better than the naive algorithm for large n, which is also the parameter range relevant for most applications.
In summary, this work is the first to apply cuckoo hashing to the problem of MPFSS and to implement a MPFSS protocol.

References

[1] *Abseil*. www.abseil.io/about. Accessed: 17.06.2019.

[2] G Amdahl. *Amdahl's law*. 1967.

[3] Sebastian Angel et al. "PIR with compressed queries and amortized query processing". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 962–979.

[4] Benny Applebaum et al. "Secure arithmetic computation with constant computational overhead". In: *Annual International Cryptology Conference*. Springer. 2017, pp. 223–254.

[5] Kenneth E Batcher. "Sorting networks and their applications". In: *Proceedings of the AFIPS Spring Joint Computing Conference*. (Apr. 30–May 2, 1969). Vol. 32. ACM. 1968, pp. 307–314.

[6] *Boost*. www.boost.org. Accessed: 09.07.2019.

[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function secret sharing". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 337–367.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function secret sharing: Improvements and extensions". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 1292–1303.

[9] Elette Boyle et al. "Compressing Vector OLE". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 896–912.

[10] J Lawrence Carter and Mark N Wegman. "Universal classes of hash functions". In: *Journal of computer and system sciences* 18.2 (1979), pp. 143–154.

[11] Hao Chen, Kim Laine, and Peter Rindal. "Fast private set intersection from homomorphic encryption". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 1243–1255.

[12] *Compute benchmark scores for Linux VMs*. www.docs.microsoft.com/en-us/azure/virtual-machines/linux/compute-benchmark-scores. Accessed: 17.07.2019.

[13] Daniel Demmler et al. "PIR-PSI: Scaling Private Contact Discovery". In: *Proceedings on Privacy Enhancing Technologies 2018 (Issue 4)*. Sciendo. 2018, pp. 159–178.

[14] Jack Doerner and Abhi Shelat. "Scaling ORAM for secure computation". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 523–535.

[15] Dimitris Fotakis et al. "Space efficient hash tables with worst case constant access time". In: *Theory of Computing Systems* 38.2 (2005), pp. 229–248.

[16] Niv Gilboa and Yuval Ishai. “Distributed point functions and their applications”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 2014, pp. 640–658.

[17] Oded Goldreich. *The Foundations of Cryptography, volume 2, chapter Encryption Schemes.* 2004.

[18] Oded Goldreich and Rafail Ostrovsky. “Software protection and simulation on oblivious RAMs”. In: *Journal of the ACM (JACM)* 43.3 (1996), pp. 431–473.

[19] John E Hopcroft and Richard M Karp. “An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs”. In: *SIAM Journal on computing* 2.4 (1973), pp. 225–231.

[20] Marcel Keller and Peter Scholl. “Efficient, oblivious data structures for MPC”. In: *International Conference on the Theory and Application of Cryptology and Information Security.* Springer. 2014, pp. 506–525.

[21] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. “More robust hashing: Cuckoo hashing with a stash”. In: *European Symposium on Algorithms.* Springer. 2008, pp. 611–622.

[22] Vladimir Kolesnikov et al. “Efficient batched oblivious PRF with applications to private set intersection”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM. 2016, pp. 818–829.

[23] Moni Naor and Benny Pinkas. “Oblivious polynomial evaluation”. In: *SIAM Journal on Computing* 35.5 (2006), pp. 1254–1281.

[24] *Obliv-C.* `www.oblivc.org`. Accessed: 17.06.2019.

[25] *OpenMP.* `www.openmp.org`. Accessed: 17.06.2019.

[26] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo hashing”. In: *Journal of Algorithms* 51.2 (2004), pp. 122–144.

[27] Benny Pinkas, Thomas Schneider, and Michael Zohner. *Scalable Private Set Intersection Based on OT Extension.* Cryptology ePrint Archive, Report 2016/930. `www.eprint.iacr.org/2016/930`. 2016.

[28] Benny Pinkas, Thomas Schneider, and Michael Zohner. “Scalable private set intersection based on ot extension”. In: *ACM Transactions on Privacy and Security (TOPS)* 21.2 (2018), p. 7.

[29] Martin Raab and Angelika Steger. “Balls into bins—A simple and tight analysis”. In: *International Workshop on Randomization and Approximation Techniques in Computer Science.* Springer. 1998, pp. 159–170.

[30] Michael O Rabin. “How To Exchange Secrets with Oblivious Transfer”. In: *IACR Cryptology ePrint Archive* 2005 (2005), p. 187.

[31] Phillipp Schoppmann et al. “Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning”. In: *IACR Cryptology ePrint Archive* (2019).

[32] Adi Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[33] Nigel P Smart. *Cryptography made simple*. Vol. 481. Springer, 2016.

[34] Herbert S Wilf. “Algorithms and complexity”. In: Internet Edition. 1994, pp. 64–65.

[35] Samee Zahur and David Evans. “Obliv-C: A Language for Extensible Data-Oblivious Computation”. In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1153.

9 Appendix

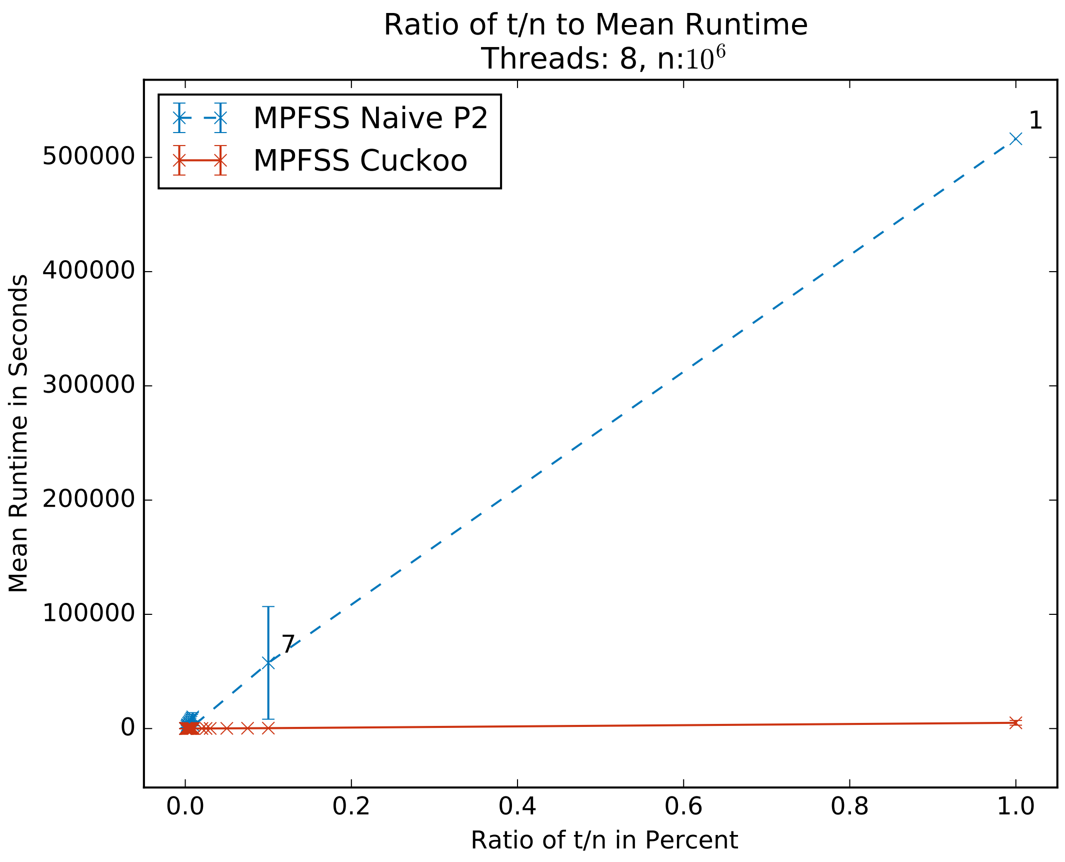


Figure 16: Runtime measurement of MPFSS Cuckoo and MPFSS Naive. Here, data from Figure 11 is plotted using a linear scale. All data points, unless annotated otherwise, consist of at least 10 measurements. Error bars represent the standard deviation. In this plot, it is clearly visible that both algorithms scale linearly but with different gradient.

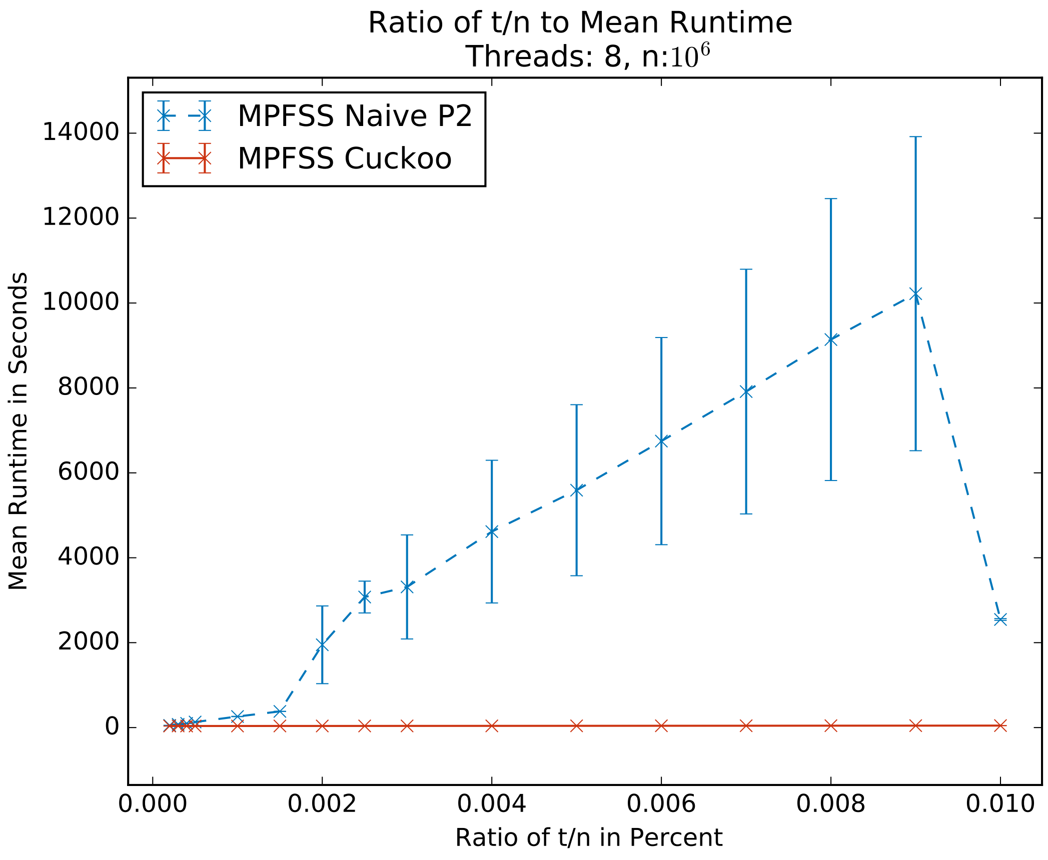


Figure 17: Runtime measurement of MPFSS Cuckoo and MPFSS Naive. This is an enlargement of Figure 16 for $5 < t < 100$. All data points consist of at least 10 measurements. Error bars represent the standard deviation.

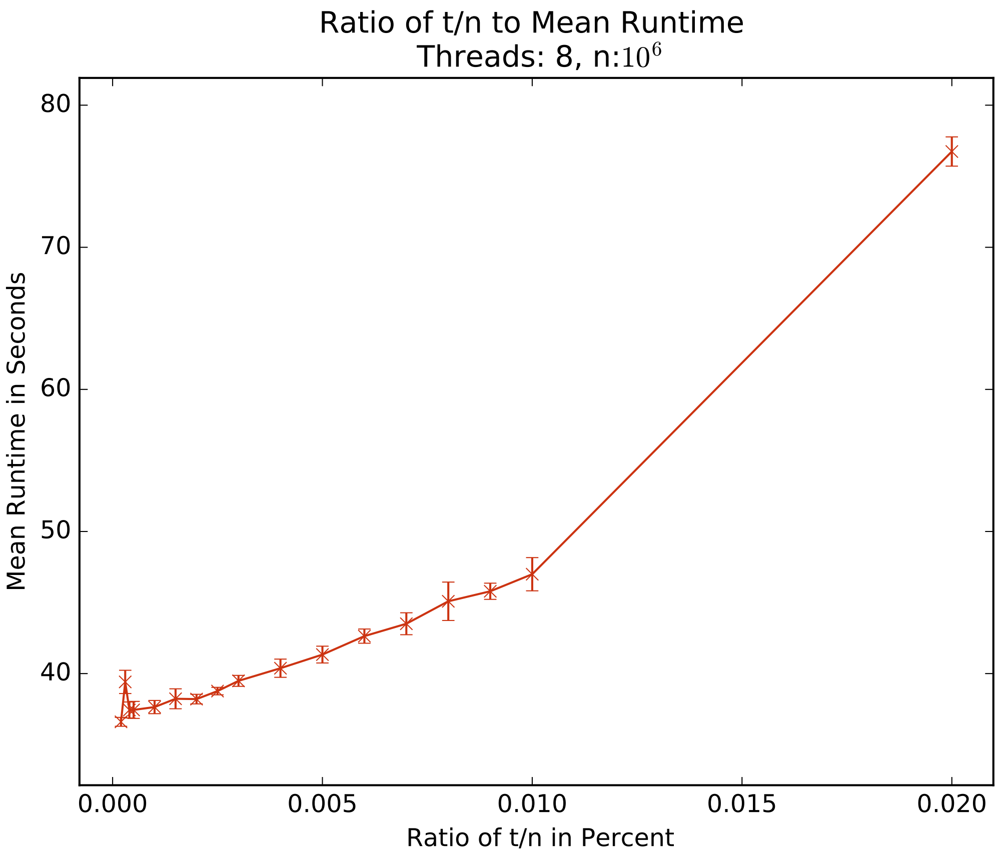


Figure 18: Enlargement of Figure 11 showing only measurement for MPFSS Cuckoo. All data points consist of at least 20 measurements. Error bars represent the standard deviation. This figure shows how for $t/n > 0.01\%$ the runtime grows linearly but with a different gradient.