Secure Datastructures based on Multiparty Computation

Tomas Toft[1]
[1]Dept. of Computer Science, Aarhus University
ttoft@cs.au.dk

Abstract

The problem of secure multiparty computation – performing some computation based on distributed, private inputs – has been studied intensively for more than twenty years. This work includes both "one shot" applications as well as reactive tasks, where the exact computation is not known in advance. We extend this line of work by asking whether it is possible to *efficiently* both update and query secret data. A clearer formulation is, perhaps, to ask whether is it possible to construct efficient datastructures based on secure multiparty computation primitives.

It is possible to construct arbitrary secure datastructures based on an oblivious RAM (ORAM). However, current state of the art information theoretically secure solutions incur a poly-logarithmic overhead on both secure computation and memory. The overhead is much smaller when considering computationally secure solutions, however, this requires secure evaluation of a one-way function as a primitive, which may reintroduce a considerable overhead.

By constructing a secure priority queue we show that practical datastructures are possible. The ideas are radically different than those used in any ORAM implementation: The present solution accesses data in a *deterministic* manner, whereas all ORAMs *randomize* the access pattern in order to hide it. The priority queue operations – insertion into the structure and deletion of the minimal element contained therein – both require $O(\log^2 n)$ invocations of the cryptographic primitives (secure arithmetic and comparison) amortized in $O(1)$ rounds amortized, where n is the overall number of operations performed.

Keywords: Secure multiparty computation; reactive functionalities; and datastructures

1 Introduction

Secure function evaluation (SFE) considers the problem of evaluating a function f on data held in a distributed manner, i.e. on $(x_1, \ldots, x_N)$ where x_i is held by party P_i. The core goal is *privacy*: to have the parties learn $f(x_1, \ldots, x_N)$, but do so without revealing any information about the x_i to anyone underway (except what can be inferred based on the output and any known inputs). The problem was originally proposed by Yao nearly thirty years ago [Yao82], and has since then been rigorously studied in the cryptographic community. The notion can be extended to secure multiparty computation (MPC), which considers reactive tasks as well. I.e. an MPC protocol may consist of multiple sequential function evaluations, where each evaluation depends on – and potentially updates – a secret state.

Depending on the desired goals, one can consider either passive or active security. In the former case, all parties follow the protocol, but some may collude, combining information, e.g. messages sent and received, in an attempt to break the privacy of other parties. In the case of active security, they not only pool their data, they may also misbehave arbitrarily. Hence protocols ensuring security against an active adversary (controlling all misbehaving parties) must be *fully robust*. The classic results of secure computation demonstrate that any function can be computed with active security and a polynomial overhead given a fully connected, synchronous network, with authenticated (and in the information theoretic (i.t.) case (where the adversary is computationally unbounded), also secure) channels, [GMW87, BGW88, CCD88].

In addition to this, many specialized protocols for specific, well-motivated problems have been proposed – auctions and data mining are two popular examples. Utilizing domain specific knowledge and focusing solely on the task at hand often allows considerable efficiency gains. However, though the solutions may require MPC rather than SFE, the tasks considered are rarely reactive themselves. Put differently, the topic of datastructures based on MPC primitives has received surprisingly little attention.

1.1 Contribution

With the exception of realizations of the oblivious RAM (ORAM; see related work below), we propose the first datastructure based on MPC. We construct an efficient priority queue (PQ) based on protocols providing secure arithmetic over a ring, $\mathbb{Z}_M$. The specifics of the underlying protocols are irrelevant, hence the construction is presented in a hybrid model, where the parties have access to secure black-box arithmetic, i.e. we assume access to secure protocols implementing simple arithmetic operations on secure data. With respect to security properties, the solution inherits those of the underlying primitives; hence to ensure security in the presence of active adversaries – i.e. to ensure that the protocol is fully robust – it suffices to require this of the underlying primitives. These exist for both the computational and the i.t. settings.

The construction is a variation of the bucket heap of Brodal et al. [BFMZ04] and allows two operations: `INSERT`(p, x) which inserts a secret[1] element, x, into the queue with secret priority p; and `GETMIN`$()$ which deletes and returns (in secret shared or encrypted form) the element with minimal priority. Both operations use $O(\log^2 n)$ primitive operations (MPC multiplications and comparisons – with the latter constructed from arithmetic) in $O(1)$ rounds, i.e. only a constant number of messages are sent per pair of parties. All measures are *amortized* – they apply on average, though individual executions may require more operations and a non-constant numbers of rounds.

[1]Secret shared or encrypted.

The overall approach taken in this paper is to construct a datastructure where the actions performed are completely independent of the inputs. From there it is merely a matter of implementing the operations using MPC primitives. This strategy presents an immediate path to the present goal, however, it is not at all clear that it is the only one, or indeed the best one.

1.2 Related Work

There are essentially three areas of related work: *incremental cryptography (IC)* due to Bellare et al. [BGG94, BGG95]; *history independent (HI) datastructures* introduced by Naor and Teague [NT01] which builds on the oblivious datastructures of Micciancio [Mic97]; and *Oblivious RAMs* proposed by Goldreich and Ostrovky [GO96].

IC considers evaluating some cryptographic function – e.g. a digital signature – on *known*, changing data *without* recomputing that function from scratch every time. HI datastructures on the other hand focus the problem of eliminating unintentional information leakages when datastructures containing *known* data are passed on to other parties. E.g. the shape of the structure itself may reveal information on the operations performed. Both consider security and structuring data, but are fundamentally different as the data is known by the structuring party.

The closest related concept is that of the ORAM. There, a CPU (with only constant size, private memory) runs some program residing in the main memory. An adversary observes the memory access pattern (but not the data/program instructions retrieved) and attempts to extract information from this. Damgård et al. have observed that (as hinted by Goldreich and Ostrovky) implementing the CPU using MPC primitives allows array indexing to be securely realized using MPC primitives, [DMN10]. Through "pointers" this allows datastructures.

Oblivious RAMs hide the access pattern by randomizing it. [GO96] achieved this through access to a random oracle, instantiated by a one-way function. This was recently improved by Pinkas and Reinman, [PR10], who brought the computational overhead down to $O(\log^2 n)$ and the memory overhead down to $O(1)$. This approach has two drawbacks: The use of one-way functions implies that the solution cannot provide information theoretic security. Moreover, the one-way function must be evaluated using MPC; this can always be done using general techniques, however, it is not clear that this will result in a practical protocol overall. Independently, Ajtai [Ajt10], and Damgård et al. [DMN10] have proposed information theoretic ORAMs. Though the solutions are different, in both cases, both the computation and memory overhead is poly-logarithmic.

Where the ORAM provides a completeness theorem, the present work focuses on whether different strategies may provide more efficient means of reaching specific goals. Indeed, the present approach is radically different than those used when constructing ORAMs: in stark contrast to the above, the access pattern of the PQ solution presented is *completely deterministic*, whereas *any* i.t. secure realization of the ORAM require at least $\log n$ bits of randomness per operation, where n is the overall size of the memory, [DMN10]. This is possible since the overall "program" (the task to be performed by a MPC protocol) is known; actions taken may therefore depend on the task at hand.

Despite the common ground, oblivious RAMs do not provide all answers regarding MPC datastructures, at least not presently. In addition to the above, instantiating an oblivious RAM with present state of the art solutions incurs at least a computational overhead of $O(\log^2 n)$ on every read/write operation – this *equals* the entire cost of our PQ operations. Moreover, the sequential nature of the ORAM implies that it cannot provide round-efficient solutions. Also, both i.t. secure ORAMs have a poly-logarithmic overhead on memory usage, whereas the present construction does not. Thus, the present work contains the *only* i.t. secure datastructure with constant memory overhead. Finally, there are no obvious reasons why the secure PQ could not be improved significantly, while an i.t. secure ORAM with constant overhead seems less plausible, hence for specific tasks it appears as if tailored solutions will always be preferable, even under big-O.

1.3 The Structure of this Paper

Section 2 introduces underlying, cryptographic primitives in the form of an ideal functionality, the arithmetic black-box. Section 3 then extends this model and introduces higher-level constructs. The problem is then formalized in Sect. 4 in the form of an ideal functionality, which is then realized in Sect. 5. Section 6 contains concluding remarks and lists open problems and possible future avenues of research.

2 The Basic Model of Secure Computation

We consider a setting where N parties $P_1, \dots, P_N$ are pairwise connected by authenticated channels in a synchronous network. They perform multiparty computation based on linear primitives; this could, for example, be additively homomorphic secret sharing or encryption. We do not specify the actual, underlying cryptographic primitives. Rather, we construct the priority queue based on an ideal functionality – the arithmetic black-box (ABB) $\mathcal{F}_{\texttt{ABB}}$ of Damgård and Nielsen [DN03] – whose definition captures the required properties. The model may essentially be viewed as if the parties have oracle access to a trusted third party providing secure storage of elements of a ring, say $\mathbb{Z}_M$, as well as providing arithmetic on the stored values. This approach hides irrelevant details of the underlying scheme, and simplifies the security proof tremendously, since an attacker can only influence the well-behaved $\mathcal{F}_{\texttt{ABB}}$ in very few ways. We stress that *any* protocols with the desired properties can be utilized, not just those of [DN03]; loosely speaking, we require only secure storage and arithmetic, and that the honest parties "agree on the state." This can be achieved in multiple ways, e.g. based on secret sharing [Sha79, BGW88] or homomorphic encryption, [Pai99, CDN01]

2.1 The Arithmetic Black-box

[DN03] presents $\mathcal{F}_{\texttt{ABB}}$ in the UC framework of Canetti [Can00] and realizes it efficiently based on Paillier encryption [Pai99], i.e. M is an RSA modulus. The protocols are shown secure against an active, adaptive adversary corrupting a minority of the parties. This implies an efficient, UC-secure realization of our protocols, as they are shown secure in the $\mathcal{F}_{\texttt{ABB}}$ hybrid model. (Note that we present a slight variation of the original $\mathcal{F}_{\texttt{ABB}}$ to better represent the required operations; indeed, the original version is quite "implementation specific.")

- **Input:** If party P_i sends "$P_i : x \leftarrow v$" and at least $\lceil (N-1)/2 \rceil$ other parties send "$P_i : x \leftarrow ?$" then $\mathcal{F}_{\texttt{ABB}}$ stores value v under the variable name x,[2] and sends "$P_i : x \leftarrow ?$" to all parties.
- **Output:** When a majority of parties provide the order "$\texttt{output}(x)$", then assuming that value v was stored under x, $\mathcal{F}_{\texttt{ABB}}$ sends "$x = v$" to all parties as well as the adversary.
- **Arithmetic:** Upon receiving "$x \leftarrow y + z$" from a majority of parties, $\mathcal{F}_{\texttt{ABB}}$ computes the sum of the values stored under y and z and stores the result as x. Similarly, upon receiving "$x \leftarrow y \cdot z$" from a majority of parties, the product is stored under x.

[2]For simplicity, consider these distinct, i.e. each value is stored indefinitely and never overwritten.

The protocols of [DN03] are quite complex, hence to explain the intuition behind $\mathcal{F}_{\texttt{ABB}}$, a simpler example is better. Consider a *passive* adversary, and MPC based on Shamir's secret sharing scheme over $\mathbb{Z}_M = \mathbb{F}_M$ for prime M [Sha79]. Secret sharing allows one party to store a value privately and robustly among multiple others. If *and only if* sufficiently many agree, the value will be revealed. Input (respectively output) simply refers to secret sharing a value (respectively reconstructing a secret shared value). Shamir's scheme is linear hence addition is simply addition of shares, while secure multiplication can be obtained through the protocols of Ben-Or et al. [BGW88].

It can be shown (given secure communication between all pairs of players, and assuming that all parties agree on the secure computation being performed) that these protocols realize $\mathcal{F}_{\texttt{ABB}}$ with perfect security in the presence of passive adversaries. Further, the protocols of [BGW88] even realize $\mathcal{F}_{\texttt{ABB}}$ in the presence of active adversaries if the corruption threshold is reduced to $N/3$.

Reducing the corruption threshold of $\mathcal{F}_{\texttt{ABB}}$ does *not* invalidate our construction. Indeed, the above presentation of $\mathcal{F}_{\texttt{ABB}}$ is a lot more rigid than required. Whether malicious adversaries are allowed, whether security is computational or information theoretical, etc, are merely details which do not affect our construction. Note further, that while the protocols [DN03] do not guarantee termination if more than $N/2$ players are corrupt, privacy is not compromised. Thus, $\mathcal{F}_{\texttt{ABB}}$ could even be modified to allow up to $N-1$ corruptions (which includes the two-party setting) at the cost of guaranteed termination. For simplicity, we retain the above definition.

2.2 Complexity

As abstract primitives are used, it is not possible to provide an exact measure of computation or communication complexity. One can merely count the number of operations performed by $\mathcal{F}_{\texttt{ABB}}$. These correspond directly to the computation and communication of the underlying primitives.

We focus on communication complexity of the operations. Recalling that we assume linear primitives, clearly this implies that addition (and multiplication by public values) is costless, as it consists of purely local computation. For simplicity, we will not distinguish between the complexities of the remaining operations, input; output; and multiplication of two unknowns. We remark that multiplication is in general the most expensive operation and also the most used one. Our construction does not require input/output, except as occurring in existing sub-protocols, whose main cost also originates from multiplications.

Regarding instantiations of $\mathcal{F}_{\texttt{ABB}}$, the basic operations are typically reasonably cheap. For passive adversaries, typically only $O(1)$ ring elements are communicated per player or pair of players. E.g. performing a passively secure multiplication of Shamir shared values can be done by having each party reshare the product of its shares (plus local computation), i.e. two field elements per pair. The dominating term of the Paillier based protocols of [DN03] – and in other actively secure constructions – is $O(N)$ *Byzantine agreements* on ring elements (e.g. encryptions) per (non-costless) operation, i.e. $O(1)$ Byzantine agreements per player. Unless a broadcast channel is assumed, such an overhead is required to guarantee robustness in the face of actively malicious adversaries.

A second measure of complexity of protocols is the sequential number of rounds required (the number of times messages are exchanged between parties). For simplicity of the presentation above, this was left out, however, it is easily incorporated into the ABB: first assume that all operations take the same, constant number of rounds. Now, rather than receiving *one* instruction from each party, the parties send *lists of independent instructions* to be performed by the functionality. Each invocation of the functionality now refers to one round of operations, which in turn translates to some number of rounds of communication. Naturally, it is unlikely that all sub-protocols require the same number of rounds, however, the approach simplifies the model at little loss.

While the straightline program notation used below improves readability, it has a drawback: The description of the protocols is detached from the actual execution in the $\mathcal{F}_{\texttt{ABB}}$ hybrid model. Hence, complexity analysis becomes slightly more complicated, as the description does not explicitly state which operations can be performed in parallel. The possibility of providing clearer descriptions easily makes up for this, though.

3 Extending the Arithmetic Black-box

The secure priority queue is not constructed directly based on $\mathcal{F}_{\texttt{ABB}}$. Instead, we extend that functionality, adding additional operations, which the parties can specify. These are realized using nothing more than the basic operations of the arithmetic black-box. This section can be viewed as containing preliminaries in the sense that it introduces a number of known constructions.

3.1 Secure Comparison

In order for the priority queue to make sense, there must be some notion of order with respect to the stored elements. Further, when $\mathcal{F}_{\texttt{ABB}}$ holds values, it must be possible to determine which is the larger. I.e. the arithmetic black-box must be extended with an additional action for this, that the parties can order it to perform.

- **Comparison:** Upon receiving "$x \leftarrow y \overset{?}{>} z$" from a majority of the parties, $\mathcal{F}_{\texttt{ABB}}$ determines if y is larger than z, and stores the result as x; 1 for true and 0 for false.

As an example, consider values taken from $\mathbb{Z}_M$ and the natural "integer" ordering. If M is prime, a secure comparison computation completely within the arithmetic black-box, e.g. [DFK$^+$06]. For this example, the best general solutions[3] known to the author requires $O(\log M)$ operations (multiplications as well as input/output) in $O(1)$ rounds, e.g. [NO07]. When M is an RSA modulus – e.g. a public Paillier key – complexity is increased to $O(N \log M)$, where N is the number of parties; this is simply due to more expensive sub-protocols. However, it is stressed that these are merely options; *any* secure computation and *any* ordering will work.

For simplicity of the present analysis, it is assumed that the comparison computation requires only a constant number of rounds. We do not specify the number of ABB-operations though. Instead, the number of comparisons used is counted separately from the basic operations due to its (in general) much higher cost. It is straightforward to multiply by the actual cost when a specific realization is given.

3.2 Secure Conditional Swap

Based on the ability to compare, it is possible to perform conditional swaps: given two values stored by $\mathcal{F}_{\texttt{ABB}}$, swap them if the latter is larger than the former. This can be viewed as sorting lists of length two, and is easily constructed within the ABB by simply computing the maximal and minimal of the two.

$$\mathit{max} \leftarrow \left(a \overset{?}{>} b\right)(a-b)+b; \quad \mathit{min} \leftarrow a+b-\mathit{max}$$

[3] In specific settings other solutions may be preferable, e.g. [Tof11].

These two expressions are easily translated to messages to be passed from the parties to the extended $\mathcal{F}_{\mathtt{ABB}}$. Clearly the whole operation requires only constant work – $O(1)$ basic operations and a single comparison – in a constant number of rounds, and naturally, multiple swaps may be executed in parallel. The swap computation can be generalized to multi-element values, say pairs consisting of a priority and a data element. It is simply a question of having a well-defined comparison operator and using its output to choose between the two candidates on a single element basis.

3.3 Secure Merging

The main, large-scale primitive of the construction is the ability to merge two sorted lists, both of some known length ℓ, stored within $\mathcal{F}_{\mathtt{ABB}}$. This will be written $\mathtt{MERGE}\,(X, Y)$, where X and Y refer to lists of stored values. A construction for this primitive is obtained from sorting networks – sorting algorithms created directly based on conditional swaps. These do not perform any branching, hence they are oblivious to the inputs – except for the problem size, ℓ – as well as being deterministic. This allows the parties to specify the relevant operations to $\mathcal{F}_{\mathtt{ABB}}$.

Any sorting network can be utilized to merge, by simply viewing the whole input as a single unsorted list. However, as the only requirement is merging, we merely take the inner workings of Batcher's odd-even mergesort [Bat68]. The whole sorting network requires $O(\ell \log^2 \ell)$ conditional swaps, however, merging alone requires only $O(\ell \log \ell)$ conditional swaps in $O(\log \ell)$ rounds. Further, the constants are low, implying that the solution is practical.

A primitive for merging lists of *differing* lengths, $\ell \neq \ell'$, is also required. To do this, the shorter list is simply padded – assume that some element, e_∞, which is greater than all others is reserved for this – such that they become of equal length. At this point the lists can be merged using the above solution. Finally, the padding must be removed. It consists of elements greater than any valid ones, hence all such elements are pushed to one side. As the size of the padding is known, those elements can be removed by truncating the list. Complexity is $O(\max(\ell \log \ell; \ell' \log \ell'))$ operations in $O(\max(\log \ell, \log \ell'))$ rounds. We overload the meaning of $\mathtt{MERGE}\,(\cdot, \cdot)$ to avoid introducing additional notation.

We present a final, needed primitive which is highly related to merging: *merge-split.* This operation, denoted $\mathtt{MERGESPLIT}\,(X, Y)$, takes two lists as input as above. As the name suggests, the goal is to merge two lists into one, which is then split (cut into two parts whose concatenation is the sorted list). The only requirement is that lengths of the new lists must equal the lengths of the old ones. The effect of a merge-split is that the most significant elements end up in one of the lists, while the least significant ones end up in the other. Naturally, both new lists are still sorted. Clearly this operation is equivalent to a merge, as the split merely renames variables. Hence, its complexity is the same as merging.

4 The Goal: a Secure Priority Queue

At this point we are ready to present the desired goal, an ideal functionality for a priority queue, $\mathcal{F}_{\mathtt{PQ}}$. However, we will not simply introduce a priority queue. The data of a datastructure is not separated from the rest of the world in general. Inputs to the datastructure do not necessarily originate at some party. It could equally well be the result of previous computation – even computation involving operations on the datastructure in question. Thus, the goal is *not* to simply construct the priority queue. The goal is to *extend* the arithmetic black-box with a priority queue.

This is done in the exact same manner as the previous section, where a comparison operation was added. Here, all operations of the datastructure in question must simply be included. Hence,

$\mathcal{F}_{\texttt{PQ}}$ contains the operations of (the extended) $\mathcal{F}_{\texttt{ABB}}$ in addition to the following two:

- INSERT(p, x):[4] Upon receiving "PQinsert(p, x)" from a majority of parties, where p and x are variables over $\mathbb{Z}_M$, $\mathcal{F}_{\texttt{PQ}}$ stores the values associated with the pair (p, x) in an internal, initially empty list, L. Finally, all parties are notified that the command has terminated.
- GETMIN(): Upon receiving "$y \leftarrow$PQgetmin()" from a majority of parties, $\mathcal{F}_{\texttt{PQ}}$ determines and deletes from L the pair in L with the lowest p-value. The corresponding x-value is stored as y, and the parties are notified that the assignment to y has been performed.

Naturally, parties engaging in a protocol may interleave these two operations arbitrarily with other computation. This could even contain operations for additional priority queues. We present only the case of a single instance for simplicity. Note, though, that the ideal functionality must treat the priority queue operations on a given PQ as atomic with respect to each other. Operations on the same queue *cannot* be interleaved. Thus, correctness is only guaranteed when all parties send the same insert/getmin order, and that order is executed *completely* before the next one is sent.

It is noted that there is a small issue with the above description: the behaviour of $\mathcal{F}_{\texttt{PQ}}$ is not specified if GETMIN() is executed when the queue is empty. In this case, $\mathcal{F}_{\texttt{PQ}}$ may simply discard the operation and do nothing. There are no consequences of doing this, as all parties always know the exact number of elements in the queue, as they are notified by $\mathcal{F}_{\texttt{PQ}}$ whenever something is inserted or deleted.

5 The Secure Bucket Heap

An standard binary heap is not directly implementable using MPC primitives. The difficulty encountered is that it is not possible to branch – one cannot traversing the tree from root to leaf by a path that depends on secret data. Instead, the efficient realization of $\mathcal{F}_{\texttt{PQ}}$ is based off of the bucket heap of Brodal et al. [BFMZ04]. However, two significant changes are made. Jumping ahead a bit, the original solution merges sorted lists in the "natural" way, i.e. using linear scans – this does not work in the present setting. However, Batcher's solution of Sect. 3.3 does; thus, all merging is performed in this manner instead. A second difference is also made to better tie in the high-level solution with the MPC primitives. This change actually causes the name *bucket heap* to be slightly misleading, as a rigid structure (with respect to the priorities) is imposed on the contents within the buckets. The name is retained, though.

The changes are made possible by considering a simpler problem than the one solved by the original bucket heap: the decrease-key operation has been eliminated. This ensures that the actual content can be ignored, while focus is kept on the *priorities* of that content. The motivation for doing this is that it eliminates the need for operations that would at best be expensive. The change also allows a clearer presentation.

5.1 The Intuition of the Secure Bucket Heap

Before going into details, the intuition behind the bucket heap is explained. The main idea is essentially to store a list, D, containing all the data in sorted order. However, doing this naively makes inserts too costly, as a newly inserted element can end up anywhere. Thus, rather than

[4]For simplicity, this will be referred to as INSERT(p) below. The element, x, is left implicit to avoid clutter.

inserting directly into that list, elements are placed in buffers until sufficiently many have arrived to pay for the combined cost of all insertions.

More formally, the data is split into sub-lists (buckets), $D_0, D_1, D_2, \ldots$, where the elements of D_i are less than those of D_{i+1}. The size of the D_i double with each step (or level) – $|D_i| = 2^i$. In addition to this, at each level, i, there is a buffer, B_i, of the same length as the data; see Fig. 1.

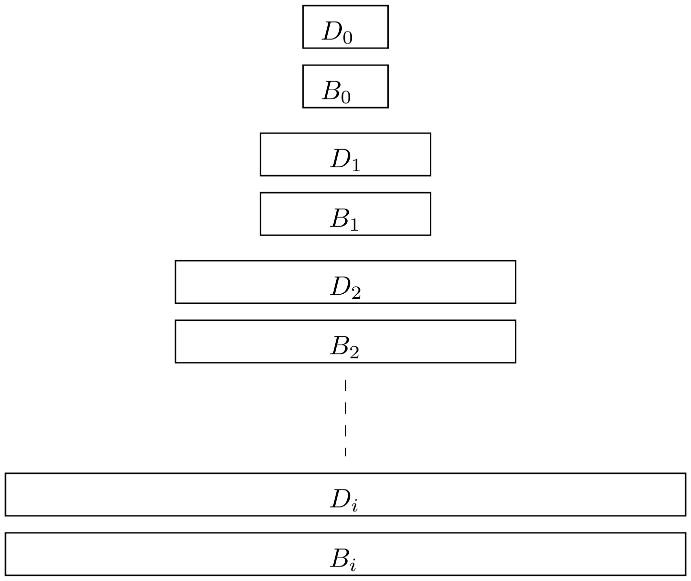

Figure 1: The structure of the bucket heap

Inserting new data means placing it in the uppermost buffer, B_0. The (very loose) intuition is now, that whenever a buffer B_i is full, its contents are processed. The elements that "belong at this level" are moved to D_i, while the rest are pushed down to B_{i+1}. The D_i can be viewed as a sorted list of "buckets" of elements, where the elements increase with each step. Thus, "belong at" intuitively means that an element is smaller than some $p \in D_i$.

The minimal is obtained by returning the contents of D_0. This may then be output by $\mathcal{F}_{\texttt{ABB}}$. For subsequent `GETMIN()`'s, D_0 may be empty. In general the desired element is in the top-most, non-empty bucket. Thus, the element is easily found, and the remainder of the content of the bucket is used to fill all the buckets above the level in question.

Strictly speaking, the above description is *not* correct at all. It does, however, explain the overall ideas quite nicely: elements are inserted in the top buffer and move downwards through the buffers until reaching the correct level (bucket). Deletions then work by pulling the contents of the buckets upwards. With this intuition in mind, we are ready to proceed to the details – the trick of the construction is to ensure that we know exactly *where* all the data is placed, while keeping secret *which* data went where.

5.2 Invariants

Data is stored as specified above, but with a few additional requirements on the data buckets, D_i, and the buffers, B_i. Bucket D_i can contain 2^i elements, and it is either completely full or completely empty, i.e. $|D_i| \in \{0, 2^i\}$. The buffers are slightly different as the B_i must contain strictly less than 2^i elements. They may temporarily exceed this limit, though – this is referred to as the buffer being full – at which point the contents will be processed. Finally, the elements of buffer B_i are greater than (have higher priority than) the elements of the higher-lying buckets, D_j, $j < i$.

In difference to the original bucket heap, the contents of the buckets and buffers are stored as lists sorted by priority. This is the rigid structure referred to above. Note that the concatenation of the D_i can be viewed as one long, sorted list.

5.3 The operations

The datastructure must be maintained using only the operations of the arithmetic black-box. There are two operations to construct, the insertion of a new value and the extraction the present minimal. The main parts of these operations are seen as Protocols 1 and 2.

The insert operation, `INSERT`(p), is performed by placing p in the top buffer, B_0. This fills it and it must be flushed using Protocol 1. The `GETMIN`() operation is realized by extracting the element stored in the top-level bucket (or at least attempting to extract; the bucket may be empty). This is done by executing `DELMIN`(0); the details are seen as Protocol 2.

Protocol 1 `FLUSH`(i) – flushing buffer B_i at level i

```
Require: Full buffer, $B_i$, at level $i$.
Operation: Flush $B_i$, moving the elements contained into data or subsequent buffers.
    if $|D_i| = 0$ and $i$ is the lowest level then
        $D_i \leftarrow B_i(1..2^i)$
        $B_i \leftarrow B_i((2^i + 1)..|B_i|)$
        if $|B_i| \geq 2^i$ then
5:          FLUSH($i$)
        end if
    else
        $(D_i, B_i) \leftarrow$ MERGESPLIT$(D_i, B_i)$
        $B_{i+1} \leftarrow$ MERGE$(B_i, B_{i+1})$
10:     Set $B_i$ empty
        if $|B_{i+1}| \geq 2^{i+1}$ then
            FLUSH($i + 1$)
        end if
    end if
```

5.4 Correctness

To show correctness, it suffices to show that the invariants hold and that these imply the desired behavior. First off, it is clear that for the starting position – an empty priority queue – all invariants hold. All buckets are empty which is acceptable; further, as there are no elements at all, the required ordering between elements of different buckets and buffers as well as the internal ordering are clearly satisfied.

An `INSERT`(p) operation places p in B_0 which is a list of length 1. Except for this full buffer, the invariant holds, as no relationship between the elements of a buffer and the ones of the associated bucket is specified. Nor is this the case for those of any buckets or buffers below. After this, the buffer is flushed.

Protocol 2 `DELMIN`(i) – return the 2^i smallest elements from level i and below (or everything if there are fewer than 2^i elements)

```
Require: Non-empty bucket heap; all levels above the i'th are completely empty.
Operation: DELMIN(i) – determine and return the $2^i$ minimal elements
     if $|D_i| = 2^i$ then
         $(D_i, B_i) \leftarrow$ MERGESPLIT $(D_i, B_i)$
         Return $D_i$ and set it empty
     else if $i$ is the lowest level then
  5:     Return $B_i$ and set it empty
     else
         $B_{i+1} \leftarrow$ MERGE $(B_i, B_{i+1})$
         Set $B_i$ empty
         if $|B_{i+1}| \geq 2^{i+1}$ then
 10:         FLUSH$(i+1)$
         end if
         $\tilde{D} \leftarrow$ DELMIN $(i+1)$
         if $|\tilde{D}| \geq 2^{i+1}$ then
             $B_{i+1} \leftarrow \tilde{D}(2^{i+1}+1..|\tilde{D}|)$
 15:         $D_i \leftarrow \tilde{D}(2^i+1..2^{i+1})$
             Return $\tilde{D}(1..2^i))$
         else
             Return $\tilde{D}$
         end if
 20: end if
```

There are two possible states when flushing a buffer, as seen from the "outer" `if`-statement of Protocol 1: either this is the lowest level and D_i is empty; *or* there is data here or below. In the first case, we may simply move the 2^i smallest elements into the data bucket (the buffer is only flushed when it is full, i.e. when it contains at least 2^i elements). As *all* the elements in the buffer are bigger than the elements in the buckets above, then the new relationship with all buckets hold.

Alternatively, there may be data in the present bucket or below the present level. By the invariant, all elements are greater than the elements of the buckets above. Thus, after performing the merge-split, line 8, all invariants still hold with respect to the levels above. The same is true for any levels below. The merge-split ensures that the smallest elements of the level end up in the bucket. These are at most as big as the previous largest element of the bucket, and must therefore be smaller than the elements of the levels below. At this point it is guaranteed that the elements of D_i are smaller than those of B_i, hence the latter can be pushed into the buffer below. All invariant still hold, except that B_{i+1} may now have become full; if so, it must be flushed

The minimal element contained in the PQ is obtained using `DELMIN`(0), which extracts the minimal from the top-most level. The intuition behind Protocol 2 is that the minimal element must come from a bucket. Only if there are *no* elements in *any* buckets can an element be taken from a buffer, line 5.

From the invariant, it is clear that the minimal element will either be in the top-most, non-empty bucket *or* a buffer above that. Hence, starting with level 0, the buffers are flushed (merged with the buffer below) one after the other until a non-empty bucket is found, lines 7 and 12. Note that this simply merges buffered elements above any full buckets, hence it does not affect the invariant.

Once a non-empty bucket is found, it is merge-split with the buffer at that level to ensure that it contains the 2^i smallest elements, not only at this level, but *overall*: buckets and buffers above are empty, and any element in the bucket is less significant than any at a level below. The present bucket is then emptied into the buckets above, which fills them and leaves one element that can be returned – this task is trivial as the elements of the bucket are sorted, and the concatenation of the buckets above should be a sorted list. It is easily verified that the invariants hold at this point.

If all buckets are empty, then all buffers are merged until only a single non-empty one exists (at the lowest level, i). Viewing B_i as a sorted list, its contents may be distributed to the top buckets above, exactly as with the emptying of a bucket above, *except* that there may be "excess elements." For $|B_i| = 2^j + k$, with $k < 2^j$, the minimal element can returned and the j top-most buckets filled. This leaves the k largest elements; these are placed in the buffer B_{j+1}. As before, the elements of the initial list, B_i, are easily distributed such that the invariant holds, as it is sorted.

5.5 Complexity

Complexity of both `INSERT`(p) and `DELMIN`(0) is $O(\log^2 n)$ amortized, where n is the overall number of operations. This follows from a coin argument, where each coin pays for a conditional swap.

When inserting an element into B_0, $\Theta(\log^2 n)$ coins are placed on it. The invariant is that every element in buffer B_i has $\Theta(((\log n) - i) \log n)$ coins, which is clearly satisfied for both the initial (empty) datastructure and for the newly inserted element. These coins will pay for the flushes caused by full buffers, the secure computation of Protocol 1.

Moving elements from the buffer to the empty bucket at the lowest level is costless. In the other case, the buffer B_i at level i must be merged with the bucket, D_i, (in the merge-split) and with the buffer below, B_{i+1}. Both merges require $O(2^i \log 2^i)$ conditional swaps – the three lists are at most a constant factor longer than 2^i. This cost can therefore be paid using $\Theta(2^i \log n)$ coins from the elements of B_i.

The merge-split potentially moves elements between the buffer and bucket, however, the number of elements in the buffer remains the same. The second merge then moves the contents of the buffer to the level below. As B_i was full, it contained at least 2^i elements, thus it suffices if each one moved pays $\Theta(\log n)$ coins. As the entire contents of the buffer is pushed one level down, the elements only require $\Theta(((\log n) - (i + 1)) \log n)$ to ensure that the invariant holds. Hence even when each element pays the $\Theta(\log n)$ coins needed for the flush, the invariant holds. Overall, this implies the stated complexity for the insert operation.

A similar argument is needed for deletion, `DELMIN`(0). However, rather than placing coins on the elements themselves, the deletion coins are placed on the buffers. Each operation places $\Theta(\log n)$ coins on each of the buffers, B_i; this requires $\Theta(\log^2 n)$ coins overall. The invariant is, that every buffer, B_i, has $\Omega(k \log n)$ coins, where k is the combined size of the empty buckets above, i.e. $k = \sum_{j=0;|D_j|=0}^{i-1} 2^j$.

Whenever `DELMIN`(i) is called, it implies that the buckets of all levels $j < i$ above are empty. Hence, the buffer B_i has $\Omega((2^i - 1) \log n)$ coins implying that it can pay for a merge at level i, either with the contents of bucket D_i or the buffer below. Either way, all buckets above are filled,[5] implying that the buffer does not need any coins to satisfy the invariant presently (except below, which has not been touched). Thus, the previous delete operations pay for the required merge.

Regarding the number of rounds, both operations require at most a constant number of merges

[5]The only possible exception occurs when all buckets are empty, but the structure is so "sparse" that not all buckets above can be filled. In this case a "completely full" structure is constructed from scratch so no coins are needed.

for each level. Hence, worst-case complexity is $O(\log^2 n)$. Amortized complexity, however, is only constant. The lower levels are rarely processed ($\Omega(2^i)$ operations occur between the ones "touching" level i) and the upper levels are cheap to process (only $O(i)$ rounds are required to merge at level i). Hence for n operations,

$$\sum_{i=0}^{\log n} \frac{n}{2^i} i^2$$

rounds are needed overall which implies $O(1)$ rounds on average.

We conclude with a sketch of the actual complexities when $\mathcal{F}_{\mathtt{ABB}}$ is instantiated. For the case of passively secure MPC from Shamir's secret sharing scheme, the $O(\log^2 n)$ conditional swaps translate into $O(\log M \cdot \log^2 n)$ basic operations of $\mathcal{F}_{\mathtt{ABB}}$, which in turn require $O(\log M \cdot \log^2 n)$ transfers of elements of $\mathbb{Z}_M$ between each pair of parties. Considering the Paillier based, actively secure protocols of [DN03], this increases to $O(N \cdot \log M \cdot \log^2 n)$ broadcasts of elements per player, where N is the overall number of players, due to the increased cost of the comparison protocol.

5.6 Security

Intuitively, security of the bucket heap follows directly from the security of $\mathcal{F}_{\mathtt{ABB}}$. An adversary, $\mathcal{A}$, can only learn information when the ideal functionality outputs a value, i.e. when the underlying primitives explicitly reveal information. However, at no point in the present computation is an output command given by any of the honest parties. Hence, as $\mathcal{A}$ does not control what amounts to a qualified set, it cannot make $\mathcal{F}_{\mathtt{ABB}}$ perform an output operation. By similar reasoning, it can be seen that no adversary – i.e. set of parties behaving incorrectly – can influence the computation resulting in incorrect values stored in $\mathcal{F}_{\mathtt{ABB}}$.

The above is of course only the intuitive explanation. Formally, the view of $\mathcal{A}$ must be simulated. The required simulator, however, is easily constructed. It simply "executes" the realizing PQ computation, except that for every operation that $\mathcal{F}_{\mathtt{ABB}}$ should be instructed to perform, the simulator will simply play the role of $\mathcal{F}_{\mathtt{ABB}}$ towards the corrupt players. It will receive their commands and send the messages (acknowledgments) to the corrupt players that they expect to receive. This is clearly indistinguishable from the point of view of any adversary. For each PQ operation, it simply sees a fixed set of messages, namely the ones corresponding to the secure computation implementing the operation, which it "*knows*" is being executed.

6 Conclusions and Open Questions

In this paper, a priority queue based on secure multiparty computation primitives was constructed. It allows a set of parties to *efficiently* maintain a set of secret values from which the minimal element is *efficiently* extractable. The amortized complexity of every operation is poly-logarithmic in the problem size. Though general solutions are possible based on the ORAM, the overhead incurred by present state of the art solutions may be prohibitive. E.g. one must either accepts a non-constant overhead on memory, or evaluate a one-way function using MPC primitives.

The present strategy for constructing the PQ was to make the operations performed oblivious to the inputs. The construction demonstrates that non-trivial, *deterministic* secure datastructures are possible. This is surprising, since accessing all data with every operation appears – at least intuitively, since branching is impossible – to be required. The term *input-independent datastructures* is coined for such constructions, and and it is noted that these can always be implemented using MPC primitives.

References

[Ajt10] M. Ajtai. Oblivious rams without cryptogrpahic assumptions. In *42nd Annual ACM Symposium on Theory of Computing*, pages 181–190. ACM Press, 2010.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[BFMZ04] G. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *SWAT*, pages 480–492, 2004.

[BGG94] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO'94*, volume 839 of *Lecture Notes in Computer Science*, pages 216–233. Springer-Verlag, Berlin, Germany, 1994.

[BGG95] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *27th Annual ACM Symposium on Theory of Computing*, pages 45–56. ACM Press, 1995.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10. ACM Press, 1988.

[Can00] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/`.

[CCD88] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19. ACM Press, 1988.

[CDN01] R. Cramer, I. Damgård, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045, pages 280–300, 2001.

[DFK$^+$06] I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryoptography*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[DMN10] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. Cryptology ePrint Archive, Report 2010/108, 2010. `http://eprint.iacr.org/`; conference version to appear at TCC 2011.

[DN03] I. Damgård and J. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer Berlin / Heidelberg, 2003.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM Press.

[GO96] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[Mic97] D. Micciancio. Oblivious data structures: Applications to cryptography. In *STOC*, pages 456–464, 1997.

[NO07] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC 2007: 10th International Workshop on Theory and Practice in Public Key Cryptography*, volume 4450 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2007.

[NT01] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *STOC*, pages 492–501, 2001.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer-Verlag, Berlin, Germany, 1999.

[PR10] B. Pinkas and T. Reinman. Oblivious RAM revisited. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 502–519. Springer, 2010.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[Tof11] T. Toft. Sub-linear, secure comparison with two non-colluding parties. To appear at PKC 2011; manuscript available from author, 2011.

[Yao82] A. Yao. Protocols for secure computations (extended abstract). In *23th Annual Symposium on Foundations of Computer Science (FOCS '82)*, pages 160–164. IEEE Computer Society Press, 1982.