Demystifying secure computation: Familiar abstractions for efficient protocols

---

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment

of the requirements for the degree

Doctor of Philosophy

by

Samee Zahur

May

2016

APPROVAL SHEET

The dissertation

is submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

AUTHOR

The dissertation has been read and approved by the examining committee:

David Evans

Advisor

Westley Weimer

Mohammad Mahmoody

Denis Nekipelov

Bryan Parno

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

May

2016

Demystifying secure computation:
Familiar abstractions for efficient protocols

A dissertation
Submitted to the department of Computer Science
Of University of Virginia
In fulfillment of the requirements
For the degree of
Doctor of Philosophy

Samee Zahur
April 2016

Abstract

Over the past few years, secure multi-party computation (MPC) has been transformed from a research tool to a practical one with numerous interesting applications in practice. MPC is a cryptographic technique that allows two or more parties to collaboratively perform a computation without revealing their own private inputs to each other (other than what can be inferred from the output result). Example uses include private auctions where all the participants keep their bids private, private aggregation of corporate-internal data for economic analysis, and private set intersection.

However, efficiency of MPC protocols have remained a persistent challenge for many applications. One particular issue that we examine in this dissertation is input-dependent memory accesses. It is difficult to efficiently access a memory location without revealing which element is being accessed, which in turn makes it very difficult to efficiently implement certain programs. This dissertation solves the problem by separately considering two different cases. First, we construct efficient circuit structures for cases where the access pattern is known to follow certain constraints, such as locality. The second case involves a new Oblivious RAM (ORAM) construction that provides general random access. The ORAM construction is slower than the specialized circuit structures, but faster than existing ORAM constructions for MPC for a large range of parameters. To help in implementing and evaluating these constructions, we also designed a new extensible programming language for MPC called Obliv-C, which we believe can be a useful contribution in its own right. We hope that these components will make it easier for programmers to write efficient MPC programs for many interesting applications.

Acknowledgments

This work was supported in part by grants from the NSF Award CNS- 1111781), the Air Force Office of Scientific Research, Google, and the UVa SEAS Teaching Internship Program.

In addition, I would like to thank William Melicher, Natnatee Dokmai, Jack Doerner and Xiao Wang for helping with the implementation of various portions of this work. Frequent discussion with Benjamin Kreuter provided valuable resources and citations that helped shape this work in its early stages. Jonathan Dorn also lent me his time with discussions and with setting up experiments. I thank abhi shelat for helping us find several further opportunities for optimizations, and thank Yan Huang, Gabriel Robins, Mohammad Mahmoody, Denis Nekipelov, Bryan Parno and Westley Weimer for their helpful comments and suggestions on this work. It was also a very rewarding experience to work with my collaborators, which includes Craig Costello, Jack Doerner, Cédric Fournet, Adrià Gascón, Jon Howell, Jonathan Katz, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, Mariana Raykova, Mike Rosulek, and Xiao Wang. And last but not the least, I would like to thank my advisor David Evans for guiding me through the process over the years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Consider the following situation: say a hospital is trying to investigate a particular symptom among its patients, and suspects that they all recently underwent a complex procedure in some other hospital. This investigation would require comparison of patient records from both hospitals, which they might be reluctant to do if there is no affiliation between them. In fact, they might be legally barred from exporting health data to another institution. This is the sort of conundrum secure multi-party computation (MPC) promises to solve.

MPC is a cryptographic technique that allows two or more parties to collaboratively perform computation on private data. In the end, the parties will learn the agreed upon outputs of the function, but nothing else about the inputs or any intermediate result. Sometimes this involves a specialized cryptographic protocol that has been hand-tuned for a given function to be computed [30, 84, 86], but generic protocols also exist that can compute any function [6, 11, 41, 106].

While these protocols have been actively studied in the literature since the 1980s [11, 41, 106], it is only in the last decade or so that these techniques have become feasible in practice [22, 24, 50, 62, 63, 73, 76, 89, 103, 110]. This is both the result of improved hardware capabilities [1, 2, 59] and improvements in the underlying protocol and cryptography [50, 61, 83, 90]. Even after a decade of excitement, however, MPC has failed to gain widespread adoption.

One of the persistent problems in writing efficient MPC programs has been the inefficiency of random memory access. Simple array accesses are extremely fast in ordinary programs, but are extremely slow in the MPC world if the access pattern also depends on secret data we are trying to hide. This, in turn, makes it harder to write efficient data structures (such as stack and queue) and write programs that depend on such data-dependent access patterns.

1.1 Thesis

We argue that new algorithms can substantially improve the efficiency of data-dependent memory accesses in MPC, which in turn improves the efficiency of many applications previously discussed in MPC literature.

1.2 Contributions

This dissertation makes three main contributions to the area of MPC. First, we designed a C-based programming language that provides fast MPC primitives. It also allows us to add new constructions to the language without having to modify the compiler each time. Second, we present constructions of various circuit structures that can efficiently perform data-dependent memory accesses when the application follows certain simple access patterns. Last, we present a new oblivious RAM (ORAM) construction that provides fast, general random access, albeit not as fast as the specialized structures. Throughout the dissertation, we show how these three components can be used together to efficiently implement various applications that have been previously proposed in the literature.

Language for oblivious computation. Even with efficient algorithms and cryptographic protocols, MPC will still remain outside mainstream use until it is packaged in a way that programmers can write their own apps in an intuitive way. To this end, we developed Obliv-C — an extension of standard C, with extra keywords to determine which variables should stay secret. It features one of the fastest runtimes for secure computation today, and won an award at the recent iDASH secure genome analysis competition in 2015 for being the fastest at a genomics challenge. Moreover, one of the reference implementations by the organizers of this competition also used Obliv-C.

The purpose of Obliv-C is to help users write efficient secure applications without cryptographic expertise, while at the same time allowing researchers to easily test out new techniques. It provides easy hooks at the protocol level that can be swapped out in case a researcher wants to test out a new technique or a protocol. If the researcher wants to test out new algorithms that are protocol-agnostic (such as our circuit structures), they should be able to write the code just once and test it on multiple protocols, with different security models. We demonstrated in this work how the Obliv-C framework allows us to implement features as simple libraries where they previously required rewriting the compiler or redesigning the language. Our framework effectively decouples the tasks of protocol design, algorithm design and application writing, so that experiments in one can be done independently of the others.

The language system itself is designed as a preprocessor on top of C, along with custom cryptographic libraries for executing protocols. This way, Obliv-C programs are compiled into plain C, which are then compiled and linked with traditional C compilation tools (e.g. GCC). This supports reusing existing tools (e.g. `valgrind`, `perf`) and system libraries (e.g. `pthreads`, `libgcrypt`) designed for C. No other framework for secure computation provides such compatibility.

Circuit structures. Even simple tasks like sorting require specialized algorithms [9, 100] in the circuit-world. Unlike regular programs, those written for secure computation cannot efficiently use features such as pointers or input-dependent memory lookups since they need a circuit representation. For instance, how would we push or pop into a stack if the parties executing the program do not even know how many items have been pushed onto it? In a given application, the number of elements in a data structure may depend crucially on input data, so we cannot reveal the element count to either party.

The underlying problem is that programs efficient in ordinary contexts can suddenly become inefficient if they are naïvely converted into a circuit for secure computation. Everyday features such as pointers and random array lookups are missing in the circuit world, and are expensive to emulate. Our approach was to devise algorithmically-generated circuit structures that can efficiently emulate array accesses at least when it is known to have certain special access patterns. For example, our stack construction has $\Theta(\log n)$ amortized access cost with very low constant overheads. These structures are made available in an intuitive library that can be dropped in as a replacement for their non-secure counterparts.

Oblivious RAM (ORAM). Other programs, however, have no such predictable access patterns to exploit. The typical solution is to use oblivious RAM, which hides the address being accessed. Previously, all implementations opted for tree-based ORAMs. They stood in contrast with older hierarchical ORAMs, which were not used because they required secure evaluation of expensive hash functions. However, previous approaches were completely unusable in practice in all but extremely long-running applications, where the high initialization cost could be amortized.

Our work revisited a particularly simple version of hierarchical ORAM, namely the square-root ORAM proposed by Goldreich and Ostrovsky [40]. We developed a way to replace expensive hash evaluations using techniques from tree-based ORAMs, but retain the extremely simple shuffle-based initializations. Chapter 4 shows this can provide bandwidth savings for as little as 144 bytes of data, and initialization cost is around 10x better over almost every size. This allows us to perform random memory accesses in secure computation fast enough to make previously intractable benchmarks like scrypt feasible.

Chapter 2

Obliv-C[1]

2.1 Introduction

A requirement for generic multi-party computation is that the program to be executed has to be represented in a *data-oblivious* fashion, where the control flow of the program does not depend on the secret program inputs in any way. Such a program can be executed on encrypted data without leaking any information about intermediate results, since the control flow is the same for all executions and does not depend on the data.

A common data-oblivious program representation is a Boolean logic circuit: every logic gate (e.g., AND, OR) is specified before the secret inputs are even known. Another popular representation uses addition or multiplication gates that operate directly on finite field elements (instead of just Boolean values). Given a circuit that describes the desired computation, the protocol specifies how to execute the circuit without revealing any inputs or intermediate results.

While many previous languages and frameworks for MPC have been developed (see Section 2.8), none are sufficiently expressive to allow programmers to implement even simple library abstractions. The reason is that these languages have been designed to provide traditional programming abstractions that hide the data-oblivious nature of secure computation from the programmer. Our approach provides high-level programming abstractions while exposing the essential data-oblivious nature of such computations.

Motivating Example. Consider this simple C example of a dynamically resized array:

```
x = ...;
DynVec *vec = dynVecNew();
for (i = 0; i < n; i++) {
    if (cond) {
        dynVecAppend(vec,x);
```

[1]This chapter is an adaptation of:
Samee Zahur and David Evans. Obliv-C: A Lightweight Compiler for Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153.

```
}
...
```

Implementing a library like this for standard computation is trivial. The DynVec object just needs to keep track of the current size of the vector, and resize an internal buffer when more space is needed to complete an operation.

Writing something similar for a data-oblivious computation, requires the compiler to implement an append under an unknown condition: the internal memory buffer must be resized regardless of the now unknown semantic value of cond, whereas the value of x should be appended into that buffer (which is now encrypted) using a conditional write that depends on the value of cond specified outside of the function.

This problem is exacerbated for more complex library abstractions. For example, an Oblivious RAM (ORAM) structure that allows random access to a memory bank without revealing anything about the access pattern. On every read or write operation it needs to do things like network transfers, pseudo-random shuffling, and cryptographic operations. Defining a simple oramWrite() function is problematic if we want to allow it to be called from inside a conditional block: the function needs to specify a whole series of operations, some of which need to be done conditionally while others are done unconditionally. Indeed, it is not clear how a traditional programming language could even be adapted to express the situations that commonly arise in data-oblivious computation.

Contributions. We show how a language can be designed to support extensible secure programming introducing control structures that expose the data-oblivious nature of secure computation. To make it easier for programmers to develop and reason about data-oblivious programs, we provide a type system that incorporates oblivious data.

Our Obliv-C language is a strict extension of C that supports all C features (including **struct**, **typedef**, pointers, recursive calls, and indirect function calls), along with new data types and control structures to support data-oblivious programs. Section 2.3 introduces our language and describes how its language constructs and type system support data-oblivious computation.

We describe the architecture of our Obliv-C compiler in Section 2.7, showing that our language can be implemented on top of a traditional language and in a way that provides high confidence that security properties of the underlying protocol are preserved.

Obliv-C is designed to enable practitioners to more easily develop scalable secure protocols, and to allow researchers to easily implement and test new features or techniques by simply writing a new libraries rather than having to modify or build a new compiler. To demonstrate how our approach supports exploration at many levels, Section 2.5 shows how Obliv-C could be used to easily implement various library-based features including range-tracked integers and multi-threading that could not be done with existing languages, and Section 2.6 shows how Obliv-C supports experimentation with protocols. Later, in Chapter 4 we also design an ORAM library written in Obliv-C. We will now provide a brief introduction to multi-party computation before delving into the description of the language Obliv-C.

2.2 Background — Multi-Party Computation (MPC)

Obliv-C is designed for writing secure multi-party computation (MPC) programs. MPC [39, 106] enables two or more parties to collaboratively evaluate a function that depends on secret inputs from all parties, while revealing nothing but only the result of the function. In most generic constructions of multi-party computation, the function to be evaluated is represented as a circuit (either Boolean or arithmetic). Numerous circuit-based multi-party computation protocols have been developed for different scenarios. In this dissertation, we focus on using our ORAM design with Yao's garbled circuit protocol. However, much of our language and algorithms will also works with other MPC protocols in general, and does not depend on any specifics of Yao's protocol.

Garbled circuits protocols involve parties, denoted the *generator* and *evaluator*. Given a publicly known function f, the generator associates each input bit with two garbled keys k^0, k^1, and computes a "garbled" circuit representation of the function f, GC_f. Given garbled keys corresponding to inputs x and y, the evaluator can obliviously evaluate GC_f to learn garbled keys for output $f(x,y)$. The generator generates and sends GC_f and the input keys for its own input. The generator and evaluator execute an oblivious transfer protocol to enable the evaluator to learn the input keys corresponding to its input without revealing that input to the generator. After obtaining its input keys, the evaluator can obliviously evaluate GC_f to obtain the output keys which are decoded in the final step.

2.3 Design of Obliv-C

Obliv-C is a strict extension of C that provides data-oblivious programming constructs. Next, we provide an overview of the design and philosophy behind the language. Section 2.3.2 presents a concrete example of an Obliv-C program. We provide details on the type system in Section 2.4. Our implementation compiles an Oliv-C program into standard C, as described in Section 2.7.

2.3.1 Overview

Obliv-C is designed to guarantee that all security properties provided by the underlying protocol are maintained, while exposing aspects of data-oblivious computation to the programmer. Our design emphases safety, guaranteeing that no information can be leaked by program executions (assuming the underlying protocol is secure) while giving programmers enough control (including the ability to circumvent type rules) to do things that would not be possible with other high-level languages.

The main construct we introduce is an oblivious conditional. For example, consider the following statement where x and y are secret data:

```
obliv if (x > y) x = y;
```

Since the truth value of the x > y condition will not be known even at runtime, this code cannot be executed normally. Instead, every assignment inside the if statement will have to use "multiplexer" circuits in much the

same way Boolean logic circuits use multiplexers to choose between two different values. We could translate this code into something like:

```
cond = (x > y); // 0 or 1
x = x + cond * (y - x);
```

This removes any explicit control flow dependency on unknown values by using conditional assignments.

Obliv-C extends C in the following ways:

- Every basic data type (e.g., **int**, **char**, etc.) has an **obliv**-qualified counterpart (e.g., **obliv int**, **obliv char**, etc.) which is represented using an encrypted value.
- Every **if** statement with a condition that depends on **obliv**-qualified data is explicitly indicated as **obliv if**. An **obliv if** statement executes in a way that prevents control dependencies from leaking the condition value.
- Type rules related to **obliv if** are enforced across function boundaries at compile time by using two different function families: ones that can be invoked from inside **obliv if**, and ones that cannot.
- Special *unconditional segments* allow library writers to perform actions unconditionally, which allow them to write various library abstractions. By being unconditional, these segments can avoid control dependecy restrictions, while executing inside an **obliv if** scope.

Next, we walk through a simple example illustrating the general structure of Obliv-C programs and how the programmer uses it.

2.3.2 Millionaires' Problem

Figure 2.1 shows an Obliv-C implementation of Yao's classic millionaires' problem [106]. It simply outputs which of two integers is greater (purportedly, to enable two millionaires to decide who should pay for dinner without disclosing their actual wealth).

When the program executes, both parties (in this protocol, although our design can support any number of parties) execute the same program. By convention, we will call them Alice (Party 1) and Bob (Party 2). The a, b, and res variables are declared using the **obliv** keyword to indicate that their values may depend on secret inputs.

The program obtains secret inputs using:

```
obliv int feedOblivInt (int value, int p)
```

This function is executed synchronously by both parties to introduce the input into **obliv int** variables of the shared computation. It converts a value from one of the parties (party p) into a new cryptographic **obliv int** value that can no longer be deciphered by either party on its own. The value provided by the other party is simply ignored. Since both parties have their own copy of each variable each party can use the myinput field

```
typedef struct {
  int myinput;
  bool result;
} ProtocolIO;

void millionaire (void *args);
```

(a) File "million.h"

```
#include <million.h>
#include <obliv.oh>

void millionaire (void *args) {
    ProtocolIO *io = args;
    obliv int a, b;
    obliv bool res = false;
    a = feedOblivInt (io->myinput, 1);
    b = feedOblivInt (io->myinput, 2);
    obliv if (a < b) res = true;
    revealOblivBool (&io->result, res, 0);
}
```

(b) File "million.oc"

```
#include <million.h>

int main (int argc, char *argv[]) {
    ProtocolDesc pd;
    ProtocolIO io;
    int p = (argv[1] == '1' ? 1 : 2);
    sscanf(argv[2], "%d", &io.myinput);
    // ... set up TCP connections

    setCurrentParty (&pd, p);
    execYaoProtocol (&pd, millionaire, &io);
    printf ("Result: %d\n", result);
    // ... cleanup
}
```

(c) File "million.c"

Figure 2.1: Code for the Millionaires' Problem.

Figure (a) shows the header file that defines the datatype, (b) describes the secure computation in a protocol-neutral manner in Obliv-C and (c) shows code in plain C that invokes the former with a specific protocol with appropriate inputs, outputs and options.

to hold their own inputs. Thus, in Figure 2.1, the first invocation of feedOblivInt() only reads Party 1's copy of myinput into the shared variable a, while the second one reads only from Party 2. These variables can still be manipulated using ordinary C operators, and even mixed with ordinary **int**s in expressions, but the results are all **obliv**-qualified and only accessible as encrypted values.

The only way any values derived from secret data can be converted back to a semantic value is by using a reveal function, such as:

void revealOblivInt(**int** ∗dest, **obliv int** src, **int** p)

When this function is invoked by both parties on the same variable src, the value is decrypted and stored into the integer pointed to by dest. If p == 0, all parties receive the result; otherwise p specifies a single party who receives it. This ensures that only the values that both parties agree to reveal are actually revealed by the execution. The underlying protocol ensures that a reveal function only succeeds if both parties provide consistent parameters to the function (e.g., it will fail if they provide different values for src or p).

To run the program, both the files in Figure 2.1 are compiled with the oblivcc command provided by our tool. It is a simple wrapper that provides a familiar command-line interface. It preprocesses any input file with an ".oc" extension to a plain C file before passing it on to gcc and links with additional runtime libraries required for Obliv-C code. Once compiled, the two parties simply execute the program with appropriate inputs like any other program: the end user does not need to know about Obliv-C or even need to install it separately.

2.4 Type System

The Obliv-C type system builds from a traditional information-flow based type system [99] with two levels of security. Variables declared using **obliv** are considered sensitive, and the type system ensures that information from these variables never flows into the non-sensitive ones through either explicit data dependencies or implicit control dependencies.

We add several rules beyond standard information-flow to support data-oblivious computation. First, we want programmers to be able to easily estimate the relative computation cost of their code, and to help programmers avoid writing unscalable code. This is why, for instance, we do not allow pointers with **obliv** addresses, or loops directly using **obliv** conditions. Obliv-C provides other means for accomplishing the same goals which make the costs more explicit and controllable.

Second, we account for the fact that control flow is not actually sensitive in our system. Any apparent control dependency indicated by our **obliv if** structures is not really a control dependency since it is implemented by converting it into a data dependency. Statements inside an **obliv if** become conditionally-executed statements that will be executed regardless of whether the controlling condition is true or false, which which have no semantic effect when the condition is false. Control flow is always public information in our system. This is what ultimately allows us to define features such as unconditional segments, which are very useful in writing libraries.

The purpose of our type rules is different from the normal purpose of information-flow type systems. The security of the **obliv** values is enforced at runtime by cryptographic means: even inspecting memory dumps or network logs should not provide any useful information. Hence, our type system is not used for preventing information leaks, it only exists to help the programmers avoid mistakes by providing compile time errors for code that would cause runtime errors or meaningless results. For example, this is legal Obliv-C code:

```
obliv int x; ...; int y = *((int *) &x);
```

Although our compiler will allow casts like this, the resulting code will not leak any information. At runtime, y will just contain gibberish bits of ciphertext. Obviously we do not recommend writing code like this, but it will not leak any information about x. The only way to reveal values is through the proper use of reveal family of functions on mutually agreed upon values.

In true C fashion, we allow programmers to shoot themselves in the foot, but provide a type system to help programmers avoid doing this accidentally.

2.4.1 Oblivious Data

The first four type rules explain how oblivious data is declared and used in programs.

Rule 1. Only basic C types (such as **obliv int**, **obliv char**, etc.) can be **obliv**-qualified. An **obliv**-qualified type represents a variable whose value may be unknown at runtime.

This excludes types such as structures, and pointers, although we do support structures with **obliv** fields or pointers to **obliv** variables. We excluded structures simply because, in our experience, it was not very common to have structures with all fields declared **obliv**. It was almost always a mix of **obliv** and non-**obliv** fields (storing e.g. sizes, counters). Functions may be qualified with **obliv**, although it has a somewhat different purpose that we will discuss in Section 2.4.3.

The following two rules provide a flow-sensitive type system that prevents sensitive data flowing into non-**obliv** variables:

Rule 2. Any expression that combines **obliv** values and non-**obliv** values results in an **obliv** value.

Rule 3. Non-**obliv** variables cannot be assigned to **obliv** values. Non-**obliv** values can be implicitly converted to **obliv** values and assigned to **obliv** variables.

The next rule limits where **obliv** values can be used, primarily to encourage programmers to avoid surprisingly expensive operations:

Rule 4. An **obliv** value may not be used as an array index, offset in pointer arithmetic, or as a shift amount in a bitwise shift expression. All other operators can freely mix both types of operands.

Note that we do allow **int**s to index into arrays of **obliv int**s, but not vice versa. Although we could have avoided Rule 4 and added support for oblivious array indexes using circuits such as full multiplexers, but

they are notoriously slow in practice. Instead, we want to encourage developers to explicitly weigh the trade-offs between various other mechanisms of indirect access, such as those using circuit structures [107] or oblivious RAM (Chapter 4), all of which can be implemented as library modules in Obliv-C. Similarly, it is a deliberate decision to not support pointers whose addresses can be unknown at runtime, or bitwise shift operators with unknown shift amounts. Such pointers would make it very easy to write inefficient programs that would need to multiplex over the entire heap at every pointer dereference.

2.4.2 Conditional Constructs

Rule 5 ensures that control flow never depends on **obliv** values, except as used in the new **obliv if** construct:

Rule 5. A condition expression of a traditional control structure (e.g. **while**, **for**, **switch**, etc.) may not be **obliv**. An **if** statement using **obliv** values must be explicitly marked as **obliv if**.

The **obliv if** statement has the following syntax:

obliv if (cond) { ... } [**else** { ... }]

Marking **obliv if** explicitly helps the programmer (and code readers), since it has implications both in the type system and in the runtime. Since the condition may not be known at runtime, *both* the consequent and alternative branches will be executed (possibly using conditional instructions) no matter what the condition actually was. As a result, execution always incurs the runtime overhead of both branches.

An **obliv if** statement introduces an **obliv** *context*, where certain operations are restricted. Non-**obliv** variables declared outside an **obliv** context cannot be modified inside it. Locally declared non-**obliv** variables, however, can be modified since they are not visible outside the **obliv** context. This allows us to run loops inside **obliv if** constructs:

```
obliv if (cond) {
    for (int i = 0; i < n; ++i) {
        // ...
    }
}
```

Without this exception for locally declared variables, we would not be able to modify i for the loop counter. But here, this is not a problem since i will go out of scope once we exit the conditional branch. Thus, this exception for locally declared variables does not violate the requirements for data obliviousness.

As we explain in Section 2.4.3, this also allows us to safely invoke functions from inside an **obliv if** even if they modify some non-**obliv** variables. Our rules for preventing such control dependencies are slightly complex since we want them to work across function boundaries, without actually inlining functions.

The restriction on oblivious values in conditional expressions for other control structures appears draconian, but is consistent with our goals to provide programmers with a clear view of the costs of different programming constructs. The amount of computational resources used by a program, such as CPU time or

memory usage, would leak information about the loop condition if the number of executions varies. Hence, loop conditions in secure programs must not depend on secret values. Instead, a data-oblivious program needs to impose a predetermined conservative upper limit to the number of iterations, and iterate that many times regardless of the condition. Within the loop body, we can use an **obliv if** statement to limit the effective number of iterations. For example, if n is an **obliv** variable, the loop:

```
for (i = 0; i < n; i++) { ... }
```

could be rewritten as:

```
for (i = 0; i < MAX_BOUND; i++) {
    obliv if (i < n) { ... }
}
```

In practice, the restriction on oblivious values in loop conditions is necessary, because whatever a loop condition is, the parties executing it will have to somehow know when to terminate the loop. Which means, it can always be written in a way such that the condition is a non-**obliv** value.

2.4.3 Functions

Not all functions can be allowed inside **obliv if**, since they may modify non-**obliv** global variables. To handle this, we introduce a second family of functions called **obliv** functions. These functions can be invoked from anywhere, but may not modify global non-**obliv** variables or invoke other non-**obliv** functions.

Here is an example of an **obliv** function:

```
void writeArray (obliv int* arr, int size,
                 obliv int index, obliv int value) obliv {
    for (int i = 0; i < size; ++i) {
        obliv if (i == index) {
            arr[i] = value;
        }
    }
}
```

The **obliv** suffix after the parameters denotes that writeArray is an can be called from inside a conditional context. The compiler checks the body of an **obliv** function indeed adheres to the restrictions on modifying global state.

As for writing to arrays at an **obliv** index, note that we cannot do much better than this in general. The standard practice is to create a linear-sized multiplexer circuit to perform the write, which is essentially what writeArray does. Each assignment inside the **obliv if** is a conditional assignment (i.e., a multiplexer between old and new values), which is controlled by a different condition for each value of i.

The type rules for **obliv** functions are:

Rule 6. Non-**obliv** functions may not be invoked from inside **obliv if** or other **obliv** functions.

Rule 7. Inside **obliv** functions, all non-**obliv** global variables are **frozen**. Moreover, they may not invoke other non-**obliv** functions.

2.4.4 Frozen State

The **frozen** qualifier allows us to safely pass variables by reference and store them in structures, as well as to reason about **obliv if** contexts more precisely.

A **frozen** variable is similar to a **const**-qualified variable. The **frozen** qualifier follows the same rules for type propagation and conversion as **const** in C. This includes the fact that a **frozen**-qualified L-value cannot be modified, as expected. In addition to the standard C rules for **const**, the meaning of **frozen** is defined by the following four rules:

Rule 8. All non-**obliv** variables defined outside an **obliv if** become **frozen**-qualified inside it (as well as in the body of the associated **else** clause). Freezing an already **frozen** variable has no effect.

Rule 9. Similarly, all non-**obliv** global variables defined outside an **obliv** function become **frozen** in the body of the function.

Rule 10. Dereferencing any pointer of type $T *$ **frozen**, for any type T, produces an L-value of type T **frozen**.

Rule 11. On **obliv** data, **frozen** qualifiers are ignored.

The reason we had to introduce a new qualifier (along with Rule 10) instead of just reusing **const** is that we frequently need to handle situations like this:

```
struct Value { int *p; } v;

obliv if (cond) { // v is frozen inside conditional context
    v->p = 5; // error
}
```

Here, if we used **const** instead, f->p would have been of type **int** * **const**, which freezes only the pointer, not the referenced value. This is not what we want, since we need all variables reachable through pointers declared outside the conditional context to be **frozen**.

2.4.5 Unconditional Blocks

Obliv-C provides a way to escape the normal type rules by using an unconditional block:

~**obliv**(*varname*) { ... }

This is only meaningful inside an **obliv if** or an **obliv** function, where code is running in a conditional context controlled by some oblivious condition. That condition is assigned to a new **obliv bool** variable named *varname*.

Code within an unconditional block may modify frozen variables:

```
typedef struct {
    obliv int* arr;
    obliv int sz;
    int maxsz;
} Resizeable;

void writeArray (Resizeable *r, obliv int index,
                 obliv int val) obliv;

// obliv function, may be called from inside obliv if
void append (Resizable *r, obliv int val) obliv {
    ~obliv(_c) { // condition unused here
        r→arr = reallocateMem (r→arr, r→maxsz + 1);
        r→maxsz++;
    }
    writeArray (r, r→sz, val);
    r→sz++;
}
```

Figure 2.2: Example use of an unconditional block.

Rule 12. All **frozen** qualifiers are ignored directly in the scope of an unconditional segment.

Code inside an unconditional block is executed unconditionally. Note that this does not risk any information leak, however, since the code in the unconditional block *always executes*, regardless of the value of the oblivious condition that would normally control its execution.

An example of its use is shown in Figure 2.2, which shows part of the implementation of a simple resizable array. It is implemented as a **struct** as shown at the top of the figure. While the current length of the array is unknown (since we might append() while inside an **obliv if**), we can still use an unconditional block to track a conservative upper bound of the length. We use this variable to allocate memory space for an extra element when it might be needed.

2.5 Extensible Data-Oblivious Programming

This section presents several examples of how the Obliv-C system supports extensible programming for data-oblivious computation. They highlight how having access to the full C language and libraries allows an Obliv-C programmer to add features to Obliv-C that would not possible in any other framework.

The first two show ways data structures can be implemented in Obliv-C that enable performance improvements that could not be done without exposing data-oblivious computation to the programmer: range-tracked integers and oblivious RAM. The next shows how programmers can incorporate special techniques into Obliv-C programs, in this case taking advantage of secret random numbers. Finally, we show how POSIX threads can be integrated into Obliv-C to produce protocols with multithreading support, demonstrating some of the advantages of seamless integration with standard C.

```
for (i = 1; i <= n1; ++i) {
  for (j = 1; j <= n2; ++j) {
    obliv int temp = omin(dp[i][j−1], dp[i−1][j]);
    obliv int d = 1;

    obliv if (temp >= dp[i−1][j−1]) {
      temp = dp[i−1][j−1];
      d = (s1[i−1] != s2[j−1]);
    }
    dp[i][j] = temp + d;
  }
}
```

(a)

```
for (i = 1; i <= n1; ++i) {
  for(j = 1; j <= n2; ++j) {
    Accum temp = acMin (dp[i][j−1], dp[i−1][j]);
    obliv bool d = true;

    obliv if (acLessEq(dp[i−1][j−1], temp)) {
      acCopy(&temp, &dp[i−1][j−1]);
      d = (s1[i−1] != s2[j−1]);
    }
    dp[i][j] = acAdd(temp, acFromBoundedOInt (0, 1, d));
  }
}
```

(b)

Figure 2.3: Computing edit distance with ordinary integers, vs. range-tracked integers

2.5.1 Range-Tracked Integers

Programs often do not need full 32-bit wide integers for all their variables, so it is possible to make arithmetic operations cheaper by using integers of limited bit-width. This can achieve significant speedups for applications that use lots of small integers, for example when counting or accumulating values. Here, we show how to write a library to support range-tracked integers that automatically maintain a conservative upper bound for a value, and resize their bit-widths accordingly.

Figure 2.3 shows an example of how it may be used. The example we use is that of computing edit distance between two strings. If the strings are of length n, we know that the results can never exceed n, and can then use appropriate widths for each integer. As shown in Table 2.1, range-tracking integers can lead to significant performance improvements.

To implement this abstraction, we define the type accum as a **struct** with fields for maintaining the actual value (which is oblivious, so not semantically known) and the conservatively-estimated maximum value:

```
typedef struct {
  obliv unsigned value;
```

	100 x 100 characters			200 x 200 characters		
	Normal int	Range-tracked	Improvement	Normal int	Range-tracked	Improvement
Total time	7.28 s	4.28 s	41.2%	23.19 s	12.04 s	48.08%
OT time	1.95 s	1.88 s	—	1.98 s	1.94 s	—
Gate execution time	5.33 s	2.40 s	55.0%	21.21 s	10.10 s	52.4%
Number of gates	1,669,010	668,429	60.0%	6,678,412	2,835,763	57.5%

Table 2.1: Improvements obtained from integer range-tracking in edit distance calculation

```
    unsigned maxValue;
} accum;
```

Note that maxValue is not **obliv**-qualified — it is a publicly known upper bound that depends on the program the two parties are executing, not on their private values. It must be public so both parties may calculate the width of the circuits needed for each operation.[2]

Here is an example how a function operating on a range-tracked accumulator could be used:

```
accum x = ...;
obliv if (y > 0) { accumAddInt(&x, 1); ... }
```

Since we expect accumAddInt to be used inside **obliv** scopes, we need to make this function an **obliv** function. Moreover, we will not know, even at runtime, if the condition $y > 0$ was actually satisfied. To hide the condition, the protocol will require executing accumAddInt() regardless of the condition.

While the implementation can conditionally modify the oblivious value, x.value, the value of x.maxValue must be conservatively adjusted regardless of the (unknown) condition. In other words, it is publicly known that the value might have increased, and so the upper bound has to increase accordingly.

Here is the implementation of accumAddInt:

```
void accumAddInt (accum *dest, int x) obliv
{
    ~obliv(en) { dest->maxValue += x; }
    int mask = (1<<width(dest->maxValue)) - 1;
    dest->value = (dest->value + x) & mask;
}
```

We use an unconditional segment to unconditionally modify the upper bound. When the actual addition is performed, we mask out the higher-order bits beyond the current maximum size to zero. This clears out any ciphertext produced for the higher order bits by the carry-out bits of addition, and allows simple bit-level constant propagation to emit fewer gates during the later arithmetic operations. Functions for min, max, addition, and copying are implemented similarly.

At this point the reader might wonder why we are implementing something so simple in a library rather than having it as built-in optimizations. Indeed, while we might add such optimizations to the compiler in the future, this example demonstrates that the programmer can go ahead and implement such optimizations as a high-level library without needing to modify the compiler. Further, even if range-tracking integers were provided by the compiler, there will always be special cases where the compiler will not be able to detect opportunities for optimization that are apparent to a programmer with understanding of deeper properties of the application. Compiler optimizations are not powerful enough to substitute for enhanced language expressiveness and control.

[2] In our implementation we also have a similar field for tracking the lower bound, but we omit that here to simplify our discussion and assume that the lower bound is always zero.

```
obliv unsigned ocRandomOblivInt(void)
{
  obliv unsigned res = 0;
  int p, pc = ocCurrentProto()->partyCount;
  unsigned x;

  gcry_randomize(&x, sizeof(x),
                 GCRY_STRONG_RANDOM);
  for (p = 1; p <= pc; ++p) {
    res ^= feedOblivInt(x,p);
  }
  return res;
}
```

Figure 2.4: Generating secret random integers.

2.5.2 Generating Secret Randomness

Generating randomness is very common operation in cryptographic protocols. There are well known examples [21] of how being able to generate secret random numbers (unknown to any party) can lead to significantly faster computation. In this section we describe how we can generate such randomness in Obliv-C and can be used as an optimization strategy.

Figure 2.4 shows a possible implementation for generating random integers. It just XORs random inputs from all parties, but does not reveal the result.

One example of its usefulness is the computation of modular inverses modulo a publicly known prime number, common in cryptography. Ordinarily, computing modular inverses require the extended Euclid's algorithm, which involves $\Theta(n)$ divisions and multiplications do be done securely in a circuit for n-bit numbers.

A faster approach would use secret randomness (similar to the techniques by Damgård et al. [21]). To compute a^{-1} mod p, we first generate a secret random number r. We then securely compute ar mod p and reveal it to everyone. Masking by a secret randomness prevents any semantic information leak.

The parties can then locally compute $x = (ar)^{-1}$ mod p, and use another secure multiplication obtain $rx = r(ar)^{-1} = a^{-1}$. Thus, we obtain the modular inverse by using just two secure multiplications and inexpensive local computation. Similar techniques can also be used to find inverses of matrices and group elements.

We ran some experiments with 32-bit integers, and found that this technique reduces runtime for inverse computation in semi-honest Yao protocols for 100 integers from 24.7 s to just 9.1 s.

This provides another demonstration of how simple Obliv-C library functions can allow users to easily write their own primitives that work seamlessly work with the rest of the language. No existing framework that provides a high-level language allows programmers to invent such primitives and perform optimizations.

Compatibility. This function would work in any protocol any protocol that supports input/output in the middle of a running protocol (e.g., semi-honest Yao as done here). However, other protocols such as the

dual-execution version of Yao will not support this because it requires all outputs to be revealed at the very end (or else it risks leaking one bit of private inputs for each round of output). If we want to help the programmer with this, for instance, it is easy to have the system refuse further computation once some output has been revealed.

2.5.3 Multithreading

Despite the prevalence of multicore processors today, no existing secure computation frameworks provide full multithreading support.[3] The reason is simply that full support requires a fairly extensive library for managing threads and providing synchronization primitives. Instead, our Obliv-C design enables users to take advantage of existing C libraries. Compared to ordinary computation, however, for threading to provide useful parallelism, two-party protocols need coordination between threads of both parties.

We implemented some threading support library to help us write the dual-execution protocol (Section 2.6.2), but we did not implement a full thread-enabled Yao yet (i.e., we have not yet implemented a user-exposed thread_create() function that can be launched during a protocol).

Implementing a protocol using multiple threads requires paying attention to three important properties, discussed below.

Network Channels. We need to set up separate TCP connections to avoid interference between data transfers for gates executing in different threads. We implemented a simple newsock=sockSplit(oldsock) function that creates a new TCP socket between parties that are already connected by an old socket. In particular, the server starts listening to a new unused port, sends the port number to the client using the old socket, after which the client connects. At this point, we can use POSIX functions to create new threads and have each thread use a different socket so that they do not interfere.

Nonces. Any gate-specific nonce value must be carefully chosen to avoid duplicates across threads. In case of Yao's protocol, this is just the gate-specific "tweak" value, or serial number used in garbling. So, for instance, if we have two threads, we should make sure that one thread is only using even numbers while the other is using odd numbers, so that they do not accidentally use the same tweak and compromise security.

Synchronization. The final point is just a general concern for all multi-threaded programs, although we should take care to use synchronization that works in a distributed fashion. While there are many synchronization primitives that are useful in programs, we just discuss mutexes as an example of how they can be wrapped for our protocols. The challenge here is to make sure that the same thread wins the lock on all relevant parties (there could be more than two in some protocols).

Figure 2.5 shows one way to implement the mutex locking function. The idea here is that only one party keeps an actual mutex, while others wait on a network signal to know that it is safe to proceed. This way, only the thread that wins the lock for party 1 will actually proceed. The unlock function simply calls

[3] There are many implementations of multiparty computation protocols that do use multithreading for executing various protocol stages [29, 53], but none of these allow application programmers to take advantage of multiple threads at the application level.

```
void obliv_mutex_lock(pthread_mutex_t* m) {
  if (ocCurrentParty() != 1) {
      recvDummy(1);
  } else {
      pthread_mutex_lock(m);
      for (int i = 2; i <= partyCount; ++i) sendDummy(i);
  }
}
```

Figure 2.5: Mutex implementation

pthread_mutex_unlock() for party 1, and does nothing for other parties. Note that this is probably not the most efficient way to implement a mutex. If thread *i* is running ahead in party 1, it will win even though other parties are still catching up. It is possible that in the meantime, some other thread became ready for all parties, and could have executed. Our proposed implementation does not take this into account, although it is possible to fix that by using another round of communication.

2.6 Implementing Protocols

So far we have focused on using Obliv-C with Yao's garbled circuits protocol for semi-honest adversaries. However, Obliv-C is designed to enable easy experimentation with any protocol that operates on individual bits for most of the computation (although other types may also be used for specific parts). This section presents two simple examples to illustrate how Obliv-C can be used to execute different protocols. Beyond these examples, there are many other protocols that could be implemented as functions for use with Obliv-C. This includes the cut-and-choose based protocols [68, 94], those in the LEGO family [28, 80], as well as those not using garbled circuits such as NNOB [79], Sharemind [15], and those based on the SPDZ family [22] (either as a full protocol restricted to Boolean gates, or as a sub-protocol for parts with many arithmetic operations). We have not yet implemented these other protocols for Obliv-C, but all of them execute in ways that fit well with our design.

2.6.1 Debugging Applications

The easiest way to discuss adding new protocols is to discuss one that performs no cryptography at all. All it does is that it provides a new function execDebugProtocol() which replaces the usual execYaoProtocol(). It simply executes the Obliv-C computation in plaintext. This speeds up the execution and makes it easier to debug Obliv-C programs. No further changes in code are necessary. After testing the program using execDebugProtocol(), we can just change that one line to execYaoProtocol() (or any other protocol launcher) to make it a secure computation.

It is easy to write new execProtocol() functions like this for launching custom protocols for use with Obliv-C. Implementing a new protocol is just a matter or defining functions for various protocol-level runtime hooks

```
void execDebugProtocol (ProtocolDesc *pd,
                        protocol_run start, void *arg)
{
  pd->currentParty = ocCurrentPartyDefault;
  pd->feedOblivInputs = dbgProtoFeedOblivInputs;
  pd->revealOblivBits = dbgProtoRevealOblivBits;
  pd->setBitAnd = dbgProtoSetBitAnd;
  pd->setBitOr = dbgProtoSetBitOr;
  pd->setBitXor = dbgProtoSetBitXor;
  pd->setBitNot = dbgProtoSetBitNot;
  pd->flipBit = dbgProtoFlipBit;
  pd->partyCount= 2;
  currentProto = pd;
  start(arg);
}
```

Figure 2.6: Implementation of the debug protocol

```
void dbgProtoSetBitAnd(ProtocolDesc* pd,
    OblivBit* dest,const OblivBit* a,const OblivBit* b)
{
  dest->value = (a->value && b->value);
}
```

Figure 2.7: Debugging protocol callback for an AND gate.

that we provide. These hooks are called do input, output, and compute a single Boolean logic gate. They simply call the user-provided Obliv-C callback function. We have already defined the various operations in terms of Boolean logic gates, so to implement a new protocol we just need to provide new implementation of these operations.

For example, Figure 2.6 shows the implementation for execDebugProtocol(). All of the first eight lines are simply setting callback functions that define various aspects of the protocol. Figure 2.7 shows how one of these callbacks could be implemented (our own implementation also keeps track of stats such as gate count etc.). OblivBit is just a C struct that represents a single **obliv bool** value. For secure computation protocols, this function would also perform other initializations like setting up pseudo-random seeds and executing base OTs. After all the initializations, the last line simply invokes the Obliv-C function provided by the user as a parameter.

We also allow developers switch out TCP/IP with their own custom transport mechanism. For example, in our experience, we often did not want to have to worry about networking issues when writing code, especially when writing a new protocol. So, when running both parties locally on the same machine, we would just pipe the data through standard input and output. In fact, even when running over a network, we can just pipe over SSH. To support this, we also provide hooks for the primitive send() and recv() functions used by various protocols, which can be replaced with arbitrary functions. This could also be used to easily inspect the

network traffic for debugging purposes or to package transmissions to improve efficiency.

Note that implementing the new protocols did not require any changes to the Obliv-C compiler. In fact, the compiler does not even need to know which protocol we are planning to execute: that can be determined later at runtime in the main() function written in C. This design makes is very easy to conduct experiments that run the same benchmark with different protocols.

2.6.2 Dual Execution Protocol

Another protocol we have implemented for Obliv-C is the *dual execution* variant of Yao's protocol [75, 105]. It provides stronger security than semi-honest versions in that it provides some guarantees even against malicious adversaries. It allows at most one bit of private data to be leaked to a malicious adversary, but requires twice the total computation since the base Yao's protocol is executed twice. Although there are even stronger protocols that provide complete privacy against malicious adversaries [41, 55, 66, 68, 79, 80, 94], they all require substantially more expensive techniques.

The basic idea for dual execution is to execute a secure computation by running Yao's garbled circuits protocol twice, but having the parties swap roles for the two executions which are run simultaneously. This way, each party gets to be the circuit generator for one execution and the evaluator for the other one. The results of the executions are tested for equality to ensure that both circuits computed the same result.

Changes to the application code needed to use dual execution are minimal. It is only necessary to swap out execYaoProtocol() with execDualexProtocol(), and have two TCP connections instead of just one, for which we provide convenient wrappers (this enables dual execution to use separate threads for circuit generation and execution that proceed in parallel).

This new function execDualexProtocol() works the same way as before, but this time it starts two threads before registering protocol-level hooks. It can now perform additional tasks like swapping roles for one thread and configuring each threat to use different TCP connections. The Obliv-C code to be executed is now launched once from each thread until it is time to perform output. During output, it needs to make sure that the output is only revealed to the evaluator side of each thread. At the same time, it accumulates a hash of the garbled wire labels, joins the two threads, performs an equality check, and returns an error to the user if the check failed.

Ideally, we want all application code to be portable across protocols. In reality, however, protocols often involve some quirks and users will have to write code carefully to achieve portability. Every protocol is expected to document its rules of usage. For example, some features like ORAMs are protocol-specific, and will not be supported in dual execution protocols. On the other hand, purely circuit-based optimizations such as integer range-tracking (Section 2.5.1) can be used with any protocol.

Other rules involve input/output timing and thread-safety. Since dual execution uses two threads, care needs to be taken when using shared memory. Dual execution has a simple restriction: the computation needs to strictly follow the "input, then compute, then output" execution model. For a semi-honest protocol, it is perfectly acceptable to reveal outputs or feed additional inputs in the middle of the protocol, interacting

with the protocol as it runs. This is not supported in the stronger protocol: in general, if we want a party to obtain an output, process it locally, and then feed it back, it is quite hard to ascertain if the data was tampered with. In theory, one could do zero knowledge proofs, but it usually is easier (and faster) to just execute the whole computation inside the secure computation protocol. Moreover, the possibility of early outputs opens the door for leaking additional information through selective failure attacks. This is a general theme for all protocols against stronger adversaries, not specific to Obliv-C, but an example of the kind of protocol-specific issue that must be adhered to when implementing applications with Obliv-C.

2.7 Implementation

The Obliv-C compiler is implemented as a modified version of CIL [78], which transforms Obliv-C code to plain C. Our source code is available under an open source license at <*http://oblivc.org/*>.

We make some changes to the CIL front-end parser to support the new language keywords and control structures. Some additional changes also were made to keep track of additional information such as the lexical depth at which a variable was declared (the default version of CIL discards this information order to simplify internal representation and processing).

Once the type-checker has completed successfully, code generation is straightforward. Figure 2.8 shows a simple example. An internal header file, "obliv_bits.h" is automatically included in the generated output files which provides the function prototypes and type declarations for the auto-generated function calls will be available during the later stages of compilation. The generated files can then be compiled normally by a standard C compiler (our oblivcc wrapper uses gcc for this).

Because of the way we implemented Obliv-C as a preprocessor on top of C, all of the normal C constructs are still available including structures, pointers, and indirect function calls. We also can trivially support separate compilation—two separate files can be independently transformed and then compiled and linked as usual. This allows us to have a feature-rich language without having to design the whole development tool chain from scratch.

Implementing obliv types. The code generator replaces **obliv** types with corresponding types that are defined as C **struct**s that represent the ciphertext for data bits, the operators get replaced with corresponding function calls. For example, the **obliv int** type is replaced with obliv_c_int which is defined as:

```
typedef struct { OblivBit bits[32]; } obliv_c_int;
```

Operations involving **obliv** types are replaced with corresponding function calls implemented by the provided library. For example, c = a + b is transformed into obliv_c_setAdd(&c, &a, &b).

Functions like obliv_c_setAdd() obliv_c_setLessThan() are defined in a runtime library that is linked with the generated C files. These functions are all defined in terms of bit operations (e.g., AND, OR, NOT). The bit operations, in turn, are implemented in some protocol-specific way, which means these back-end functions are usually written in plain C. To change the protocol, all we need to do is provide new implementations of

```
void millionaire (void *args) {
    ProtocolIO *io = args;
    obliv int a, b;
    obliv bool res = false;

    a = feedOblivInt(io->myinput, 1);
    b = feedOblivInt(io->myinput, 2);

    obliv if (a < b) res = true;

    revealOblivBool(&io->result, res, 0);
}
```

(a)

```
void millionaire (void *args) {
    ProtocolIO *io = args;
    obliv_c_int a, b;
    obliv_c_bool res;
    memset (&a, 0, sizeof(obliv_c_int));
    memset (&b, 0, sizeof(obliv_c_int));
    memset (&res, 0, sizeof(obliv_c_bool));

    a = feedOblivInt(io->myinput, 1);
    b = feedOblivInt(io->myinput, 2);

    obliv_c_bool cond;
    obliv_c_setLessThan(&cond, &a, &b);
    obliv_c_condAssign(&cond, &res, &obliv_c_true);

    revealOblivBool(&io->result, res, 0);
}
```

(b)

Figure 2.8: Transformation of Obliv-C code. Obliv-C code for the millionaires' problem, before and after it is transformed to plain C by our compiler (reformatted for readability).

these operations (Section 2.6 presents an example).

Transforming conditional code. Code generation is done differently inside an **obliv if** or **obliv** function, since all assignments now must be done conditionally. To ensure that uninitialized garbage values to not interfere with conditional assignments, all local **obliv** variables are initialized to zero.

Nested **if** conditions are handled by AND-ing the new condition with the current, enclosing one. Whenever an **obliv** function is called, the current condition simply gets passed in as a hidden parameter, so that the function can continue to perform proper conditional assignments. When an **obliv** function is called outside of any **obliv** scope (that is, not under the control of any condition), the hidden parameter is just set to true, effectively making it unconditional. This is why **obliv** functions and non-**obliv** functions have different signatures in our language: internally, they accept different parameters. Similarly, Obliv-C supports two flavors of function pointers corresponding to these two flavors of functions. Thus, this transformation eventually removes all control dependencies related to **obliv if** structures.

None of these transformations interfere with the usual control structures of C (**if**, **for**, **while**, etc.). All behave as expected without any transformation. For example,

```
obliv if (cond) writeArray (arr, size, index);
```

is compiled to:

```
writeArray (cond, arr, size, index);
```

Something more complicated like:

```
obliv if (x < y) {
    for (int i = 0; i < n; ++i) {
        if (i % 2 == 0) {
            a[i] = b[i];
        }
    }
}
```

compiles to:

```
obliv_c_setLessThan (&cond, &x, &y);

for (int i = 0; i < n; ++i) {
    if (i % 2 == 0) {
        obliv_c_condAssign (&cond, &a[i], &b[i]);
    }
}
```

Note that the conditional assignment is needed only for **obliv** variables and ++i did not need any change. This works because any code that attempts to make problematic modifications to non-**obliv** variables inside an **obliv** scope will be rejected in our type-checking phase. Moreover, the conditional assignment only uses the

conditions of enclosing **obliv if**s. We do not need to separately account for non-**obliv** conditions like `i < n` or `i % 2 == 0` since those control structures are not oblivious and will execute normally.

Since loops and function calls remain in code as is, we never need to unroll or inline them into full circuits for execution, unlike other systems [48, 73]. Hence, we can run programs involving billions of gates without worrying about running out of memory. Memory management is not different in our system, since we still have full access to the usual C runtime library functions (although sometimes protocol-specific restrictions can apply, as seen in Section 2.6).

The last new feature we need to support is unconditional segments. Code written inside such a segment is simply rewritten as if it appeared outside any conditional context. Inside an unconditional segment, all code is executed unconditionally. Before this block is executed, however, the new variable of type **obliv bool** is simply initialized with a copy of the current condition so that it is available to the code in the body of this segment.

Security argument. Our design makes it easy to provide a strong argument that an Obliv-C program never leaks any secret information (so long as the underlying secure computation protocol is secure). Since **obliv** variables are encrypted data, there is no risk that they will be leaked or used in a way that leads to an implicit leak since the semantic value is not even visible to the executing program. The only way a semantic value is produced is through a call to a `reveal()` function that can convert from **obliv** variables to the non-**obliv** ones.

The code generator never generates a `reveal()` function, except where the corresponding function was used in the input program. So, we can never accidentally leak information from **obliv** variable if the type system is flawed. An error in the type system can result in incorrect code and surprising behavior, but never an information leak. For example, if the type system mistakenly allows an externally visible non-**obliv** variables to be modified in an **obliv if**, the resulting program would modify the variable regardless of the **obliv** condition (without branching). This emphasizes that our system relies on cryptography at runtime to provide security; the type rules are designed only to prevent programming mistakes.

2.8 Related Work

Many frameworks for secure computation have been published in recent years. Broadly speaking, they can be classified into two categories. First is the family of low-level frameworks that provide a library of cryptographic primitives that can be used to develop arbitrary protocols. Examples include FastGC [50], SCAPI [24], and L1 [92]. The advantage of using these frameworks is that they provide a high degree of customizability over the actual protocol execution. On the downside, however, users are generally expected to be experts either in cryptography, or in circuit structures, or both. The frameworks provide little or no type safety to prevent semantic errors, and it is difficult (or in some cases, impossible) to write applications in a way that it is portable across different protocols. In comparison, applications programmed in Obliv-C are fully portable across all protocols that work on Boolean circuits (unless they are written to deliberately use protocol-specific extensions). Moreover, the Obliv-C type system prevents accidental mistakes on the part of

the programmer, without being so restrictive that it prevents programmers from writing useful functions.

The second family of frameworks entail high-level languages that try to completely abstract away the cryptographic parts, and allow the user to code in a special language as if it was ordinary programming. Examples include Fairplay [73], CMBC-GC [48], KSS [63], PCF [62], Wysteria [89] and PICCO [110]. Unlike Obliv-C, these languages provide little opportunity for users to extend or alter protocols short of modifying the compiler directly. For example, none of these would allow a user to write custom ORAM protocols (since they manage all network traffic) or implement custom data structure libraries (since they manage all memory allocation) as we demonstrated was straightforward with Obliv-C. Some like Wysteria, though, provides very strong static type system that we do not — our type system is only intended to prevent mistakes, and relies on the underlying cryptography for security.

Thus, we consider Obliv-C to be somewhere in between the two previous families of secure computation frameworks, obtaining the best of both worlds. It provides sufficient control to enable rich extensibility, without requiring a programmer to design low-level circuits or understand the underlying cryptography.

Although our current implementation provides fairly good performance, it still does not incorporate all the optimizations that have been proposed recently. This includes using AES-NI instructions [63, 90] to garble each gate, or OT-extension for malicious adversaries [54] (our current dual execution implementation does not use OT-extension). The design of Obliv-C makes it easy to incorporate those optimizations, and any newly discovered ones, without making any changes to the compiler.

Holzer et al. [48] attempted to leverage C in secure computation, but did not support most of the C language, while Obliv-C is a strict extension of C. Finally, since their approach generates a full circuit representation before actually executing it, it cannot scale to large circuits.

Finally, there are many other implementations that use a custom designed intermediate language to address memory issues such as PAL [76] and PCF [62]. These frameworks do not support custom sub-protocols the way we do. In this respect, they are closer to the other high-level languages that we have mentioned previously, since they abstract away the data-oblivious nature of computation and provide something closer to ordinary computation. Without a full static type system, they had to take draconian measures such as not allowing function calls within an if statement that depends on secret input, for example. This greatly limits the general applicability of these systems, and requires programmers to build applications in unnatural and tool-specific ways.

2.9 Conclusion

Multi-party secure computation is a vibrant and rapidly advancing research area, but progress is impeded by the difficulty in experimenting with protocols, applications, and implementation techniques with current systems. Researchers with new ideas for implementing secure computation protocols, or for optimizing applications, tend to find it necessary to implement a new protocol from basic primitives since previous frameworks lack the necessary expressiveness to experiment with new ideas at multiple levels of abstraction.

Obliv-C provides an extensible programming tool for secure computation that provides a new option by exposing the important aspects of data-oblivious computation, while providing a high-level language and the ability to seamlessly integrate with standard C code.

Chapter 3

Circuit Structures[4]

3.1 Introduction

Generic secure computation protocols and symbolic execution both require arbitrary algorithms to be converted into static circuits, and their efficiency depends critically on the size of the circuit. Therefore, we can improve the speed of these applications by finding efficient circuit constructions for various common programming constructs.

We show efficient constructions for three common data structures: stacks, queues and associative maps. Our constructions are general enough to be used in both the applications. Our stack and queue provide conditional update operations using only amortized $\Theta(\log n)$ gates for each operation, while associative map uses amortized $\Theta(\log^2 n)$ gates for each access or update (where n is the maximum number of elements in the structure). We then show how various common array usage patterns can be rewritten using these data structures, thus obtaining far more efficient circuits for those cases (Section 3.4). Finally, we demonstrate that the use of these circuits indeed leads to significant speedups in practice (Section 3.6). We do this by manually replacing standard arrays with our circuit structures in various interesting applications of secure computation and automatic test-input generation (Section 3.5).

In the next section, we present the motivation for our work emphasizing the commonality of static circuits across applications, followed by background on how programs are typically converted into circuits. Section 3.7 discusses related work more broadly.

[4]This chapter is an adaptation of:
Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In 34th IEEE Symposium on Security and Privacy, 2013, San Fransisco.

3.2 Motivation

There has been a long history of work designing efficient hardware implementations of Boolean circuits, starting with Shannon's work in the 1930s [93]. Hardware circuit designers typically have to worry about circuit depth, gate delay, and power consumption, but view reuse as a design goal. Circuits used in several security applications are quite different. In these applications, each wire in the circuit holds a constant value during the entire execution. This is essential for privacy for secure computation applications, and necessary for test input generation where the goal is to find inputs that lead to a particular output. We call such static-value circuits, *static circuits*. For most applications, including the ones we focus on here, the cost of evaluating a static circuit follows directly from the number of gates in the circuit.

Static circuit structures are radically different from typical hardware circuits. A hardware circuit for adding a million integers, for instance, can fetch them one-by-one from memory, accumulating the sum using a single two-integer adder circuit. Describing the same computation with a static circuit requires a giant structure that includes a million copies of the adder circuit. One particular problem that stems from this difference is that random array access is horrendously expensive in static circuits. Each access of an n-element array requires a circuit of $\Theta(n)$ size where the entire array is multiplexed for the required element by the index being accessed (Figure 3.1). If the array access is performed in a loop, the corresponding circuit blows up in size extremely rapidly since static circuits cannot be reused. Of course, there are simple cases where this is not a problem, particularly when the access is at positions known in advance (e.g., Figure 3.2). It is often not the case, however, that all access positions can be determined without knowing the input data. We concentrate on making efficient circuits for the cases in between these extremes: where we know that the array is accessed in some simple pattern, but the indices do depend to some degree on the input data. The overall insight is that most programs do not access arrays in ways that require the general linear multiplexer structure because the actual array accesses are limited in predictable ways. Here, we show how to amortize the cost of multiple accesses when the application either makes multiple accesses that can be performed in a batch, or has some locality in the indices accessed.

In the following subsections, we describe our two target applications: generic secure computation protocols, and automated test-input generation using SAT solvers. We describe their typical use cases, their current state of the art, and how these applications depend on static circuits. Both applications require arbitrary programs to be expressed as static circuits, so efficient circuit constructions yield immediate efficiency gains for both applications.

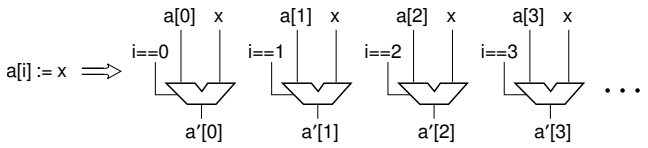


Figure 3.1: A single array access requiring n multiplexers.

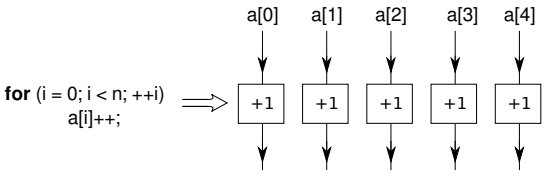


Figure 3.2: Input-independent memory accesses. The index value becomes plain constants once the loop is unrolled. Since the index does not depend on unknown inputs, array access is much cheaper.

3.2.1 Generic Protocols for Secure Computation

Secure computation allows two (or more) parties to compute a function that depends on private inputs from both parties without revealing anything about either party's private inputs to the other participant (other than what can be inferred from the function output). While there are many application-specific protocols for performing specific tasks securely [30, 77], recent advances in generic protocols enable arbitrary algorithms to be performed as secure computations. To execute any given program securely under such a protocol, it is first converted into a Boolean circuit representing the same computation. After this, the generic protocols specify mechanical ways in which any circuit can be converted into a protocol between parties to perform the same computation securely. The fastest such protocol currently known is Yao's garbled circuits protocol [67, 106]. Recent implementations have demonstrated its practicality for many interesting applications including secure auctions [16, 60], fingerprint matching [52], financial data aggregation [15], data-mining [84], approximate string comparison, and privacy-preserving AES encryption [50].

The static circuits needed for these secure computation protocols do not support fast random access to array elements. This is inherent, since the circuit must be constructed before the index being accessed is known. Any arbitrary array access requires a $\Theta(n)$-sized multiplexer circuit in the general case. While recent work by Dov Gordon et al. [45] has improved the situation for large arrays by with a hybrid protocol using oblivious RAM (ORAM), that approach still has a very high overhead. On the other hand, our approach can be orders of magnitude faster whenever it is applicable, which covers many common cases. (Section 3.7 provides a more detailed discussion.)

Other generic secure computation protocols such as fully homomorphic encryption [33, 98], GMW [41], and NNOB [79] also use static circuit representations of the computation. Therefore our circuit structures are useful in all such protocols, although we only consider garbled circuits in our evaluation.

3.2.2 Symbolic Execution on Programs

Another common application of static circuits is in symbolic program execution. Several recent works use symbolic execution to automatically derive properties about program behavior [20, 58, 88]. Several tools are able to analyze legacy programs without requiring any modification to their source code [18, 19, 37].

The particular use of symbolic execution that we consider is automatic test-input generation. The goal here is to analyze a given program and automatically come up with input cases that would drive the program

execution along a given path. By exploring all paths to find ones that end in "bad" program states (e.g., a crash or buffer overflow), these tools either obtain concrete test cases that expose program bugs or provide assurance that no such bad paths exist (at least within the explored space).

Test-input generation works by first converting the relevant part of the program into a query for a constraint solver (such as Z3 [23] or STP [32]). This solver is then used to solve for the inputs that will drive program execution to the desired state (or undesired state, as the case may be). Rapid advances in heuristic solvers over the last decade have made it possible to use these tools in many interesting new applications. It turns out that these queries are also equivalent to static circuits [97] in the sense that they also define relationships between variables in a program. Thus, if we can create optimized circuits for programs we also speed up test-input generation, increasing the scale and depth of programs that can be explored.

Since the literature in symbolic execution typically does not refer to circuits, but instead talks solely in terms of constraints, we clarify the relationship between them with an example. Consider the code fragment:

$$x := 5 + y;\ \textbf{if}\ (x > a)\ \textbf{then}\ x := x/a;$$

The goal of the symbolic execution is to check if a division by zero can arise for any particular values of y and a. Normally, the code path to division would be translated into the following constraints: $x = 5 + y$, $x > a$, $a = 0$. If all these constraints can be satisfied for some x, y, and a, we have a possible bug. Solving the constraint is done by feeding it into a SAT solver (as is often needed). The addition and greater-than operations need to be defined using primitive Boolean gates such as AND, OR, etc. much the same way hardware logic gates are used to form addition and comparison circuits. So, whenever we say that the "wires" for p and q are fed into an AND gate to produce the output wire r, what we really mean is that we are adding a new constraint of the form $r = p \wedge q$. This in turn gets translated into $(p \vee \neg r) \wedge (q \vee \neg r) \wedge (\neg p \vee \neg q \vee r)$ which is the input to the SAT solver. Thus, our optimized circuit constructions will be used to produce smaller constraint sets for encoding various programming constructs.

Since arrays can rapidly drive up costs, SMT solvers used in test-input generation tend to put a lot of effort into handling them properly. Some approaches rely on complicated under-approximation strategies that that use a simplified, but less accurate, circuit for quickly discarding obviously unreachable code paths. More accurate circuits are then generated only for the remaining paths. In our evaluation, we do not use any such approximations — instead we generate completely faithful circuits and show how they can be optimized in various cases. We hope that this will enable faster generation of test-inputs by allowing SMT solvers to use fewer, simpler approximation circuits, thereby completing analysis using fewer invocations of computationally expensive SAT solvers. We discuss this further in Section 3.4.2.

An important characteristic of constraint solvers is that they support cyclic circuits. In the end, their input is just a set of logical constraints on a set of variables. Hence, it is perfectly acceptable to have constraints such as $a = b \vee c$ and $c = a \wedge \neg d$ even though that may seem like circular definition — it is just a set of constraints on the values of the variables a, b, c and d. We will see later that this allows us to optimize random array access in the general case, something we could not do in the case of secure computation.

3.3 Background

When programs are compiled into static circuits, conversion for most simple statements and conditionals is fairly intuitive. First, statements such as $x := x + 5$ are converted into single assignment form $x_2 = x_1 + 5$, so that each variable is assigned a value only once. This way, we can now allocate separate wires in the circuit to represent the values of x before (x_1) and after (x_2) the assignment.

Conditional branches are done by separately converting each branch into a circuit. At the end of the branch, any variable modified along either path is multiplexed at the end according to the branch condition. For example,

if (a == 0) x = x + 5

is converted to

$x_2 = x_1 + 5;\ x_3 = \text{mux}(a == 0, x_1, x_2);$

where mux(p, a, b) uses its first argument as control bits to select between its second and third arguments.

Such multiplexers are actually never emitted by older symbolic execution tools, since they only explore a program one path at a time. However, this often led them to face the exponential path explosion problem. Modern tools, therefore, often explore paths at the same time using techniques such as path joining [46, 65] and compositional symbolic execution [4], which require such multiplexers (the literature also refers to them as *if-then-else* clauses).

Loops and Functions. Loops and functions pose particular challenges for static circuits. Loops are completely unrolled for some number of iterations, and functions are entirely inlined (see Section 3.7 for details). Since all values are static, and we cannot reuse the same loop body circuit for different iterations. Instead, we instantiate many copies of the same circuit. In test-input generation, there are a wide variety of heuristics for determining an unrolling threshold, which can be as simple as always unrolling just once. In such cases, checking tools simply do not explore paths that require multiple loop iterations, ignoring bugs that depend on such paths [8, 27]. Finding loop bounds is orthogonal to our work, and we do not address it here. In our evaluation, however, we use programs with known loop bounds, and those loops are completely unrolled when converted to circuits.

In secure computation, the practice is a little different. The loop bounds are known based on the given input data size even before the computation begins. An upper bound to input data sizes is publicly revealed even though the data itself is private. For this, they describe circuits in custom languages [47, 73, 92] where inputs known at circuit generation time are treated differently from inputs to the circuit wires itself. Even if the computation is written in a traditional language such as C [48], constraints are placed on what kinds of variables can define loop bounds.

```
if (x != 0) {                    t = stk.top();
  a[i] += 3;                     t += 3;
  if (a[i] > 10) i++;     ⟹     stk.condModifyTop (x != 0, t);
  a[i] = 5;                      stk.condPush (x != 0 && t > 10, NULL);
}                                stk.condModifyTop (x != 0, 5);
```

Figure 3.3: Using stacks instead of arrays. This is possible when program changes i only in small increments. We assumed that the stack top has already moved to the position corresponding to a[i] using condPush during previous increments.

3.4 Circuit Structures

In this section we present circuit structures that provide efficient constructions for stacks, queues and associative maps. The stack and queue have quite similar structures, and are therefore discussed together. In each case, we discuss the operations we support, the circuit size required for each, and when these more efficient constructions can be used in place of general arrays.

3.4.1 Stack and Queue

We replace arrays with stacks and queues whenever we can determine that the array index only changes in small increments or decrements. In other words, we use them to exploit locality whenever possible. Figure 3.3 shows an example of code transformations required for this. The operations needed to support the transformation are simply *conditional* variants of standard stack and queue operations. Each conditional operation takes an extra Boolean input that either enables or disables the corresponding modification to the stack. So, for example, stk.condPush (c,v) would push v onto the stack if c is **true**. Otherwise, the stack passes through unmodified.

Therefore, we now need to implement such operations efficiently. The operations we support for stack are condPush, condPop, condModifyTop, and readTop (no conditional read is needed since it has no side effects). The queue operations are identical, except that we use the word Front instead of Top. Figure 3.4 shows a naive implementation of the condPush operation, which still suffers from the expected $\Theta(n)$ cost per operation. We first describe the efficient circuits for stacks; then, we summarize the differences for queues.

In terms of array access patterns, we can use these two structures to optimize any case where the index moves in small increments or decrements. For example, if it is only incremented (or only decremented) in a

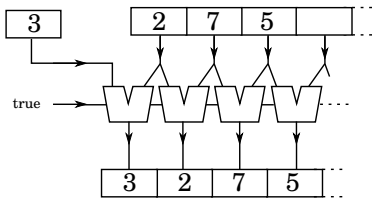


Figure 3.4: A naïve circuit for condPush, using a series of multiplexers. Since the condition is **true**, a new element is pushed. Had it been **false**, the stack would passed through unmodified.

code fragment, we use the pattern in Figure 3.3. If it moves in both directions, we just need two stacks "head-to-head", so that a pop from one is matched by a push into the other. If we need multiple array indices, we can use queues. For example, if i and j are both scanning through the same array, the space between them can be modeled as a queue, while the other segments of the array can still be stacks. This can be generalized to multiple index variables in the obvious way by using multiple queues for parts between any two consecutive index variables.

Hierarchical Stack Implementation. The key idea is to split up the stack buffer into several pieces and have empty spaces in each of those, so that we do not have to slide the entire buffer on each operation. This is illustrated in Figure 3.5. This idea was inspired by the "circular shifts" idea described in Pippenger and Fischer's classic paper on oblivious Turing machines [87]. However, our construction, which we describe now, is significantly modified for our purposes since we do not want to incur the overhead of a general circuit simulating an entire Turing machine. Section 3.7 explains the differences between Pippenger and Fischer's construction and ours in more detail.

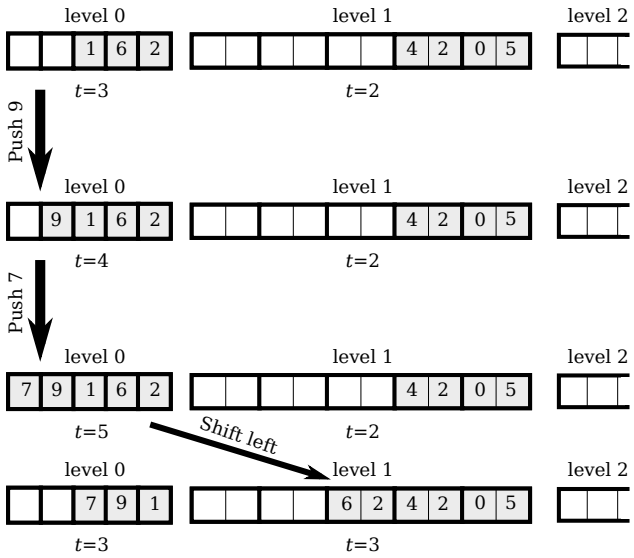


Figure 3.5: The stack buffers separated into levels, with five blocks each. A level 0 shift circuit is generated after every two conditional push operations.

The buffers of the stack are now in chunks of increasing size, starting with size 5, and then 10, 20, 40 etc. In general, the buffer at level-i has 5×2^i data slots, where the levels are numbered from the top starting at 0. The left side in the figure at level-0 represents the top of the stack. We also maintain the invariant that the number of data slots actually used in the buffer at level-i is always a multiple of 2^i. So, for example, the level-1 buffer only accepts data in *blocks* of two data items. To keep track of the next empty block available, we also maintain a 3-bit counter at each level, t. At any given state, the counter can have values in the range 0–5, and if the one at level-i reads p, then it means that the next data block pushed in to this buffer should

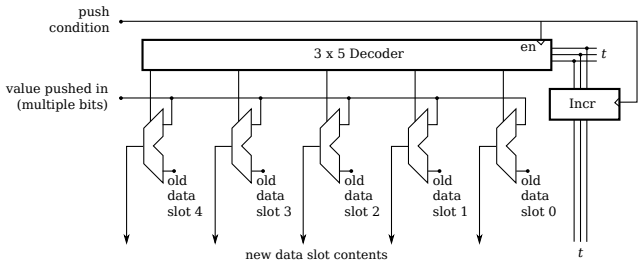


Figure 3.6: Circuit for a single conditional push operation into level-0 buffer.

go to position p. Finally, we maintain a single-bit "present" flag associated with each data block to indicate whether or not it is currently empty. We do not explicitly show this bit in our figures, other than indicating it with the color of the data block (gray indicates occupied).

Conditional Push Operations. For simplicity, let us start with a "nice" state, where we assume that the counter at each level starts no higher than 3. The top row in Figure 3.5 depicts such a state. This means every level currently has enough empty slots to accept at least two more blocks of data. So, for the first two condPush operations, we know we have an empty space at level-0 to store the new element as needed if the input condition is **true**. This circuit will simply be the naïve array write operation. In addition, we conditionally increment the counter t, to reflect the change in position of the stack top. This circuit is a series of 5 multiplexers, each of which chooses between the new incoming data being pushed on the stack and the old data stored in the corresponding slot at level-0. The multiplexer control lines are the outputs of a decoder driven by the counter, so that only the appropriate data slot gets written to, while the other items pass through with their values unchanged. Finally, since we are implementing a conditional push, the input condition feeds into the enable bit of the decoder, and conditionally increments the counter at level 0. The wires for the deeper levels (not shown in the figure) are passed through unchanged to the output. This construction is shown in Figure 3.6.

After two conditional push operations it is possible that the level 0 buffer is now full, and we have to generate some extra circuitry for passing blocks into the next level. For this, we simply check the counter to see if it is greater than 3. The result of this comparison is used to conditionally right-shift the contents of the buffer by 2 slots, while the elements ejected from the right are pushed into level 1 by a conditional push circuit for the deeper level. At the same time, the counter on buffer 0 is decremented by 2 if necessary. Of course, if the counter is already less than or equal to 3 (e.g., if the previous two conditional push operations had false conditions and did not actually do anything), the stack state is not modified in any way and the circuit simply passes on the current values (Figure 3.7). After all this is done, we can now be sure that the level-0 buffer once again has at least two empty slots for the next two push operations to succeed. The circuit for conditional push into deeper levels is the same as the one for level 0, except that the circuit that shifts to level-i is only generated every 2^i condPush operations.

Conditional Pop. The conditional pop circuit is designed analogously to the conditional push. Normally, it

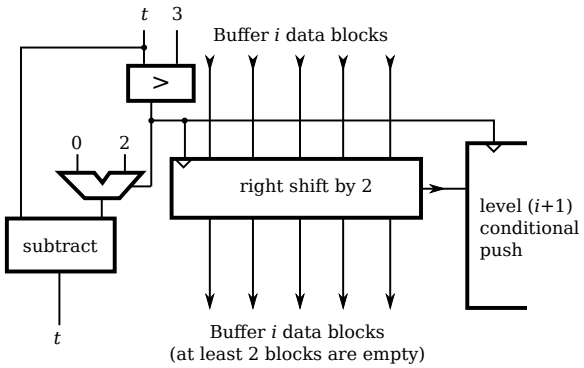


Figure 3.7: Emptying out data blocks from level i to $(i+1)$.

reads from the level-0 buffer and then decrements t. After every 2^i conditional pop operations we add extra circuits to check if the counter at level i is less than 2, and if so, pop 2^{i+1} items from the deeper level (that is, from level $(i+1)$ to level i). Whenever we pop from level i, we decrement the counter at that level, and add 2 to the counter at level $(i-1)$ (unless $i-1<0$). That way, the topmost item on the stack is always in the level-0 buffer. The read top and modify top operations, therefore, just need to use the level-0 counter to determine which of the five elements in the buffer is actually the top, and requires a constant-sized circuit, such as a multiplexer. In this case we do not even need to check all 5 data slots, since some are always kept empty by construction (they are used only in transient states just before a shift).

Since we want to support push and pop operations interleaved arbitrarily, we have to make sure that e.g., a shift from level 0 to level 1 after some conditional push still leaves enough elements at level 0 for any subsequent pops, so that they do not interfere with each other. We only shift two blocks at a time, and only do this when $t \geq 4$, so that after a shift we still have at least two blocks left for any pops that may follow. Similar logic also holds for shift after pop, when we must leave enough empty spaces for push operations, while populating this level for pops. This also explains our choice of using 5 blocks on each level: in the worst case we need 2 empty spaces for push, 2 filled blocks for pop, and one extra space since the deeper levels only take an even number of blocks.

Analysis. From this point we will use the term *cost* of a circuit to mean the number of gates. This is the most important metric for our target applications, and the depth of the circuit is mostly unimportant.

For every two push operations of the stack, a level-1 push circuit is generated only once. This will in turn cause a level-2 push circuit on every four operations of the stack, and so on. In general, for a finite-length stack known to have at most n elements at any given time, k operations access level i at most $\lfloor k/2^i \rfloor$ times. However, the operations at the deeper levels are also more expensive since they move around larger data blocks — each circuit at level i has $\Theta(2^i)$ logic gates per operation. So when k is large, we generate $\Theta(2^i \times k/2^i) = \Theta(k)$-sized circuits at level i. And since we have $\Theta(\log n)$ levels, the total circuit size for k operations is $\Theta(k \log n)$. Thus, the amortized code for each conditional stack operation is $\Theta(\log n)$. If no

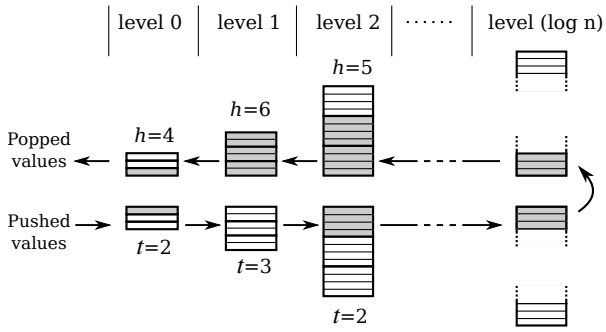


Figure 3.8: Hierarchical queue construction.

upper-bound to the stack length is known in advance (at circuit generation time), we can simply assume that after k conditional push operations, the length is at most k. Following similar reasoning, the amortized cost of a conditional pop is also $\Theta(\log n)$. Operations condModifyTop and readTop only involve the level-0 buffer, and therefore have fixed costs.

Hierarchical Queue Implementation. The queue structure is essentially equivalent to a push-only stack and a pop-only stack, juxtaposed together (Figure 3.8). Each uses 3×2^i data slots at level i, exactly half the buffer we have that level. The head and tail buffers individually are smaller than in the case of stack (3×2^i instead of 5×2^i) since we know it is either push-only or pop-only. Every level now has two counters, one for head and the other for tail. Each is represented by three bits, with values in the range 0–6, representing the head and tail position of the queue in the current buffer, respectively. If their values are h and t in the buffer for level i, it represents the fact that buffer slots $2^i t, 2^i t + 1, \ldots, 2^i h - 1$ are currently occupied. The invariant $t \leq h$ is always maintained. If $t = h$, it represents the condition where the corresponding buffer is empty. In such cases, we will always reset t and h to the value 3, so that they both point to the middle of the buffer. Here, we are using the convention that pop operations occur at the head, while push operations occur at the tail.

Conditional push and pop operations still work at level-0 as in the stack. After every 2^i push operations, we check level-i and shift two blocks to level-$(i+1)$ if $t < 2$. Similarly, after 2^i pop operations, we resupply the head buffer with new elements from the next level if $h < 5$. So far, this is exactly the same scenario as in the stack. But we now need to add some extra circuitry to transfer elements between the two halves. In particular, when a level-i pop occurs, it is possible that level-$(i+1)$ is empty, or that it does not even exist (that is, we have no wires representing that buffer). So we need to add extra circuits to check for this case. When it occurs, the next few pop operations will supply data straight out of the level-i tail buffer (instead of popping from the empty buffer at the deeper level). Similarly, after a level-i push, if the tail buffer is getting full and the next level is empty, the circuit also checks to see if the head buffer in the current level is also empty. If so, it simply shifts data blocks from the tail buffer directly to the head buffer in the same level, skipping the next-level buffer. All these conditional data movements add extra multiplexers, but increase the

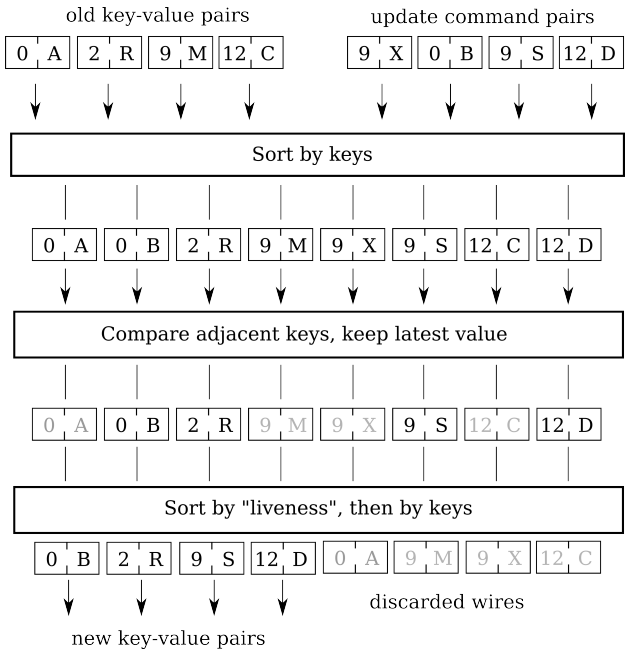


Figure 3.9: Circuit for batch-updating an associative key-value map.

circuit-size only by a constant factor. So, we can still provide conditional queue push and pop operations at $\Theta(\log n)$ cost. As we will see later in our evaluation, the constant factors are still quite small, and the total cost of our queue construction is only slightly higher than that of our stack.

Finally, this design ensures that the queue head is always found at the level-0 buffer. So, the read/modify operations for the queue head can still be done at constant cost.

3.4.2 Associative Map

The circuits for associative maps are quite different since in this case we cannot rely on any locality of access. Instead, here we amortize the cost whenever we have multiple (read or write) operations that can be performed in a "batch". The only constraint here is that none of the keys or values used in batched operations may depend on the result of another operation in the same batch since this would lead to a cyclic circuit. Many applications do have batchable sequences of array accesses, such as those that involve counting or permuting of array elements (see Section 3.6 for examples). We start with the construction for performing batch writes on an associative map, and then show how it can be tweaked to perform other operations. Here, we define an *associative map* in a circuit as simply a collection of wires representing a set of key-value pairs where the keys all have unique values. We support batched update and batched lookup operations; inserting new values can be done simply by performing updates on non-existent keys.

The circuit for performing a batch of update operations is shown in Figure 3.9. The circuit takes in

two sets of inputs: the old key-value pairs and a sequence of write operations. The write operations in the sequence are themselves also represented as key-value pairs: the key to update and the new value to be written. However, the sequence can have duplicate keys, and the order of writes among those with the same key matters. The output of the circuit is simply the new key-value pairs for the map. The idea is inspired by the set-intersection circuit used by Huang et al. [49]. We first perform a stable sort on all the key-value pairs, which is then passed through a linear scan that marks for removal all but the latest value for each key. Finally, another sort operation is performed, but this time with a different comparison function — this allows us to collect together only the values not marked for removal.

The cost of this cicuit is just the cost of two sorting operations plus a simple linear scan in the middle. Empirically, we found that it works best when the batch size is between approximately n and $2.5n$, where n is the number of key-value pairs in the map. If the batch size gets larger, we can split it up into smaller batches. If the batch is too small, we can still use this method, but the amortized cost may be higher in that case.

The sort operations each require $\Theta(n \log n)$ comparisons, and the linear scan requires $\Theta(n)$ operations. So, the circuit size should be simply $\Theta(n \log n)$[5], providing n operations each with an amortized cost of $\Theta(\log n)$. However, there is a problem. We need an oblivious sorting algorithm, where compare and swaps are hardcoded in the circuit. But, the known efficient oblivious sorting algorithms [9, 42] are not stable — they do not preserve the order of elements in input that compare equal. So to make the sorting stable, we need to pad each element with extra wires feeding them with their sequence number in the input ordering, so that even equal elements no longer compare equal during the sort. The downside of this is that we added a $\Theta(\log n)$ cost to our comparison functions, increasing our amortized cost to $\Theta(\log^2 n)$ per write operation. Obviously, this is not necessary if, in our application, we know all the keys in the write are unique. In that case the entire operation is reduced to the simple union operation for associative maps commonly found in many programming languages.

It is now easy to see how this technique can be used to perform other operations. For example, if we wanted to add to old values instead of overwriting them (e.g., if the values are integers), we just need to replace the linear scan in the middle. Even better, since addition is commutative and associative, we do not need a stable sort, allowing us to construct the complete circuit with just $\Theta(n \log n)$ gates.

If we want to perform read operations, the input will be the map key-value pairs and keys to be read. Assuming we have k keys to be read, they are all padded with sequential serial numbers $0, 1, \ldots, k-1$, which will be used later in the final sorting operation to order the output wires. But for now, they are all sorted by just the keys as before. The next step will now be filling in values for the requested keys, with a very similar linear scan. Finally, a sorting step reorders the values and presents them in the order in which they were requested (by comparing the extra serial numbers initially padded in), so that we know which output wire corresponds to which requested key. All this requires $\Theta(n \log^2 n)$ logic gates.

In the case of automatic test-input generation, one special observation is that cyclic circuits are allowed in its constraint sets (Section 3.2.2). So we can actually have circuits where some of the input keys or values

[5] As we note in Section 3.5, more popular $\Theta(n \log^2 n)$ algorithms actually perform faster for the small data sizes used in our evaluation

depend on some of the output wires of the circuit. This allows us to represent arbitrary random array access such as where one write depends on a previous read. In fact, it is quite easy to make small tweaks in the structure described to make a single, unified (but more expensive) circuit that accepts an arbitrary mix of read and write commands to be applied in sequence. Since each circuit can do n operations using just $\Theta(n \log^2 n)$ gates, the amortized cost for each array access now becomes just $\Theta(\log^2 n)$.

3.5 Implementation

Figure 3.10 depicts the system we use to implement and evaluate our circuit structures. We start with a program, which is then manually converted into the corresponding circuit generator. For this step, we use a custom circuit component library written in Haskell that includes all our data structures (Figure 3.11). This circuit generator is then executed to produce either a description of a secure computation protocol or a SAT query for test-input generation. For secure computation, we generate circuits in the intermediate representation designed by Melicher et al. [74]. For test-input generation, we generate standard DIMACS queries that is accepted by nearly all SAT solvers. We used a fast and popular off-the-shelf solver called Lingeling [12]. The following paragraphs describe various details of the implementation that efficiently realize the designs presented in the previous section. Our implementation and code for all the data structures presented here is available for download from http://mightbeevil.org/netlist/.

Multiplexing in Stacks and Queues. Since most of the data movement in the stack and queue circuits is done by generating multiplexers, they are the most expensive parts of our circuit. Hence, we focused on reducing the number of multiplexers needed.

Consider the conditional shift operations used to move data blocks between consecutive buffer levels. Figure 3.12 (a) shows what happens when a shift occurs after a pop. Observe that the leftmost two data blocks do not change regardless of whether shifting actually occurred or not. Thus, the input wires can just pass through for these blocks without needing any muxers. We take advantage of similar opportunities for the right shift needed after push operations (Figure 3.12 (b)), when the output is always a blank block.

This provides substantial benefits by itself, but also enables further reductions that take advantage of knowing blocks are blank at circuit generation time. Figure 3.13 shows how we can reduce a wide multiplexer

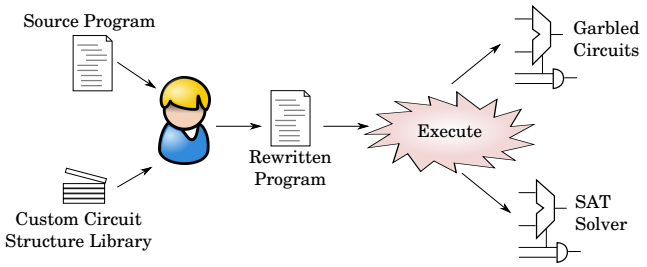


Figure 3.10: System for testing circuit efficiency.

```
if (x != 0) {
   a[i] += 3;
   if (a[i] > 10) i++;
   a[i] = 5;
}
```

⟹

```
t = stk.top();
t += 3;
stk.condModifyTop (x != 0, t);
stk.condPush (x != 0 && t > 10, NULL);
stk.condModifyTop (x != 0, 5);
```

⇓

```
   t ← top stk
   t ← add t (constInt 3)
xnz ← netNot =<< equal x (constInt 0)
stk ← condModTop xnz t stk
 c2 ← netAnd xnz =<< greaterThan t (constInt 10)
stk ← condPush c2 (constInt 0) stk
stk ← condModTop xnz (constInt 5) stk
   ...
```

Figure 3.11: Steps needed to convert code to circuit. First we simply replace arrays with appropriate data structures whenever possible (Section 3.4). Then everything is systematically replaced with library calls for generating corresponding circuits e.g. '+' becomes 'add'. Our custom library automatically handles everything internal to the data structure (e.g., condPush automatically decides if it also needs to perform an internal shift etc.). Both these steps are currently done manually (and often combined into a single step).

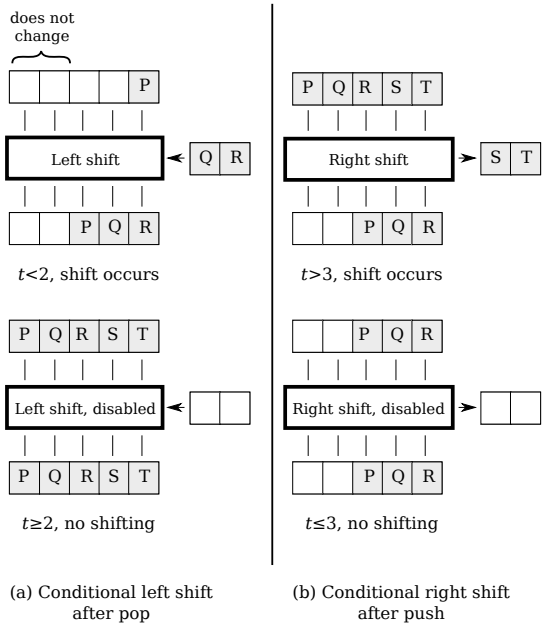


Figure 3.12: Removing muxes for shifting by determining outputs when generating circuits.

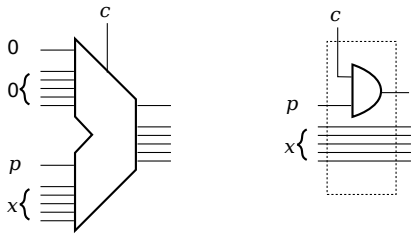


Figure 3.13: Reducing multiplexers to a single AND gate if one input is known to be blank. If the "present" flag associated with a data block is 0, then the value in the data wires is never used. So, if the blank option is selected by the control bit c, we zero out just the present flag. Note that simple constant propagation would have produced an AND gate for every output wire.

into a simple 1-bit AND gate if we know that one of the input data blocks is empty. In secure computation, we have to be careful not to do this if the wires immediately lead to the final output, since this may reveal extra information. Inside our stack and queue constructions, though, this is not a problem. Together, these techniques produce about 28% improvement for the stack circuits, while a more modest 12% for the queue.

Sorting for Associative Maps. Our associative maps require keeping the elements in key-sorted order using a *data-oblivious* sorting algorithm — one where the order of comparison and exchange operations does not depend on the actual values being sorted, since the circuits must be static. Standard sorting algorithms (e.g., Quicksort) are not data-oblivious since the comparisons they do depend on the data. Goodrich's data-oblivious randomized Shellsort algorithm [42] requires $\Theta(n \log n)$ compare and exchange operations. However, the classical algorithm of Batcher's odd-even mergesort [9] produces smaller circuits when we have fewer than about 300,000 elements[6] in the array, even though the latter algorithm has $\Theta(n \log^2 n)$ complexity. All the array sizes we use in our evaluation are small enough for Batcher's algorithm, which is the one we use.

We also take advantage of knowledge about which parts of the input are already known to be sorted. For example, if the associative map is being used as an array, the old data elements (top-left of Figure 3.9) are already sorted by their index. So, we only sort the command part, and then merge the two sorted parts in a single merge operation. This reduces the overall cost of the batch operation by another 20%. Furthermore, during array operations the keys associated with the old values are just sequential integers that are known at compile time, so basic constant propagation provides another small speedup.

3.6 Evaluation

Our evaluation is divided into three parts: first, we compare the size of our three circuit structures with the ones used in practice for various data lengths; next, we use them in garbled circuits and measure the

[6]The threshold apparently rises to 1.2 million elements when performing secure computation in the fully malicious model, since using randomized Shellsort then requires an extra shuffle network. Thanks to abhi shelat for pointing this out.

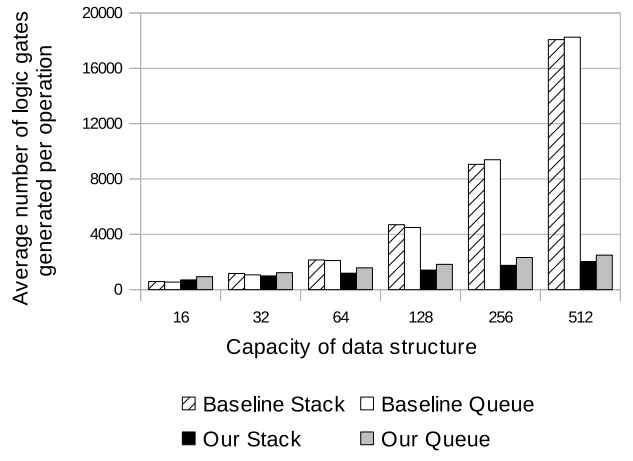


Figure 3.14: Stack/queue circuit size. The x-axis shows the maximum capacity of each data structure in number of 16-bit elements. The y-axis is the number of gates.

protocol execution time; finally, we show improvements in test-input generation by automatically producing an input that exposes a buffer overflow bug in an example function. For both secure computation and test-input generation, we were able to obtain at least an order of magnitude speedup for large cases. All the timing measurements were made on a desktop machine running Ubuntu 12.04 on an Intel i7 2600S CPU at 2.8 GHz with 8 GB of memory.

3.6.1 Circuit Size Comparison

The graphs in Figure 3.14 show how the size of our stack and queue circuits scale with increasing data sizes. We report the total number of binary logic gates used.[7] The baseline structures that we compare against are implemented using conventional circuits whose cost scales as $\Theta(n)$ for each operation. We made simple optimizations to the baseline implementation to provide a fair comparison. For example, for the first few operations of a stack, we know that the top of a stack can lie within a small range of indices, and therefore require smaller multiplexers. The stack and queue circuits were generated using random push and pop operations. As expected, the stack and the queue structures have very similar characteristics, and we reduced circuit size by over 11 times for 512 elements. Thus, when converting programs to circuits, these structures can easily replace arrays even for small sizes whenever the access pattern permits.

Figure 3.15 shows the benefits of our associative map construction. The baseline in the figure shows the cost of a normal read access by using a multiplexer. We compare that against the size of a batch read circuit on an integer-to-integer associative map. Our structures are worse than the baseline implementation for small sizes, but become beneficial for modest sizes. For 1000-element arrays, our associative map design reduces

[7]The literature in secure computation often excludes XOR gates in circuit sizes since many protocols, including garbled circuits [61], can be implemented in ways that enable XOR to be computed without any cryptographic operations or communication. While we do not show separate graphs plotting only non-XOR binary gates, we note that they show very similar trends — the y-axes simply gets scaled across the board by a factor of approximately one-third.

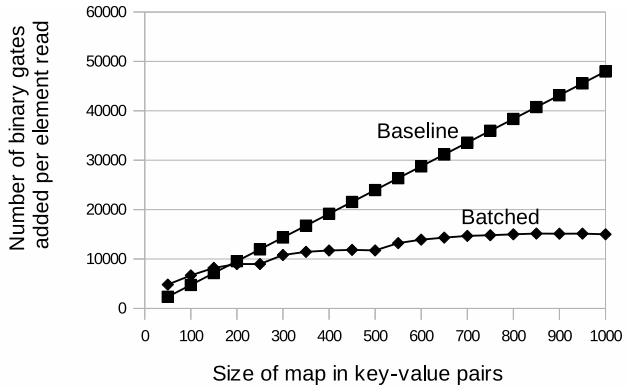


Figure 3.15: Batched read circuit size. Per-element cost of performing n read operations on an array of n elements. Baseline uses a simple multiplexer, while the "batched" circuit uses an associative map. All maps are integer-to-integer maps, values being 16-bit integers and keys $(\log_2 n)$-bit integers.

the circuit size by $3.2x$. We performed similar experiments for write operations and integer add operations. The trends are quite similar, although batched integer add has much smaller circuits since stable sorting is not required (Section 3.4.2).

3.6.2 Secure Computation Using Garbled Circuits

Here we demonstrate how much speedup our circuit structures can provide in a garbled circuits protocol execution. For this, we use two simple example programs that we executed in garbled circuits: a simple statistical aggregation program, and a data clustering algorithm.

Histogram of Sums

Consider a scenario where companies want to aggregate financial data and generate a report that provides a broad picture of the economy that all the companies can use to make better decisions. However, such data is obviously considered sensitive, and nobody wants to reveal their data to a consortium member who might be from a rival company. This is an example where secure multi-party computation was actually used in practice, as described by Bogdanov et al. [15]. Instead of directly sending their data, they send out cryptographic shares of the data to multiple servers which are then aggregated securely to form a report.

We consider one simple example of such an analysis: histogram generation. Suppose the numbers in the actual data have each been divided into two additive shares A and B, such that the i^{th} data element is simply the sum $A[i]+B[i]$. All we want to do now is compute the frequency of each data element inside a secure computation protocol so that no party learns the actual unaggregated data. The code is shown in Figure 3.16.

To execute this algorithm as a garbled circuits protocol, the code first needs to be converted into a circuit. In our case, both the arrays are of equal length n. Once the loop is unrolled n times to be converted into a circuit, it is easy to see that i values will all become constants that do not depend on input data. So the only

Input: $A[i]$ 1st share of i-th data item
Input: $B[i]$ 2nd share of i-th data item
Output: $H[d]$, frequency of d in data set
1: $H \leftarrow \emptyset$
2: **for** $i \leftarrow 1, n$ **do**
3: $\quad H[A[i]+B[i]] \leftarrow 1 + H[A[i]+B[i]]$

Figure 3.16: A simple aggregation of data from secret shares for computing histograms. Output is a simple frequency distribution of the component-wise sums. All elements are 16-bit integers, with sums being modulo-2^{16}.

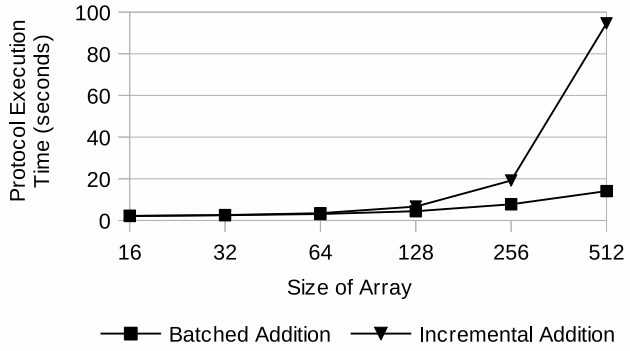


Figure 3.17: Execution time for the histogram-of-sums protocol for financial data aggregation. All inputs are 16-bit numbers.

array access that will be slow is the one on line 3 — the sums are not known ahead of time and depend on input values. Since we are performing addition on the elements of H, we can take advantage of our batch element addition circuit here. The speedup we achieve just by making this one single change is shown in Figure 3.17. For the largest test cases we tried (with $n = 512$), we reduced runtime from 1 minute 18 seconds to just under 7 seconds — a $6.7x$ speedup.

DBSCAN Clustering

Clustering algorithms are often used to uncover new patterns in a given database. For instance, insurance companies could perform clustering to find out how many demographic categories they have in their customer base in order to offer insurance plans accordingly. While it is common for a single company to perform such analyses on their own database, companies might sometimes want a more complete picture by collaborating and running a clustering analysis on their combined databases. Directly sharing such data, however, may not be desirable with rivals, or even possible because of their customer agreements. Thus, it would be desirable to perform this over a secure computation protocol. Here we will show how our stack structure can help construct efficient circuits for a popular such clustering algorithm, DBSCAN [25].

The input is simply an array of multi-dimensional data points. Some of its elements come from one

Input: P: an array of data points
Input: $minpts, radius$
Output: $cluster$: cluster assignment for each point
Output: c: number of clusters

```
1:  n ← |P|
2:  c ← 0
3:  s ← emptyStack
4:  cluster ← [0,0,...]
5:  for i ← [1,n] do                                              ▷ (A)
6:      if cluster[i] ≠ 0 then
7:          continue
8:      V ← getNeighbors(i,P,minpts,radius)
9:      if count(V) < minpts then
10:         continue
11:     c ← c+1                                                   ▷ Start a new cluster
12:     for j ← [1,n] do
13:         if V[j] = true ∧ cluster[j] ≠ 0 then
14:             cluster[j] ← c
15:             s.push(j)
16:     while s ≠ ∅ do                                            ▷ (B)
17:         k ← s.pop()
18:         V ← getNeighbors(k,P,minpts,radius)
19:         if count(V) < minpts then
20:             continue
21:         for j ← [1,n] do                                      ▷ (C)
22:             if V[j] = true ∧ cluster[j] ≠ 0 then
23:                 cluster[j] ← c
24:                 s.push(j)
```

Figure 3.18: DBSCAN implementation.

```
while c1 do
    {loopBody1}
    while c2 do
        {loopBody2}

            ⇓

selector ← outerLoop
while selector ≠ outerLoop ∨ c1 do
    if selector = outerLoop then
        {loopBody1}
    if c2 then
        selector ← innerLoop
        {loopBody2}
    else
        selector ← outerLoop
```

Figure 3.19: Flattening two nested loops into one. This produces smaller unrolled circuits if a strong bound can be obtained for the total number of iterations of the inner loop.

party, while the rest from the other. The output is the number of clusters found, and optionally, the cluster assignment of each data point (where $cluster[i] = j$ iff the i^{th} data point was assigned to cluster j). This assignment could then either be directly revealed to the respective parties, or be used in further computation (e.g. to calculate the size, centroid, or variance of each cluster).

The DBSCAN algorithm, shown in Figure 3.18, is a density-based clustering algorithm that runs a recursive search through the input data set for regions of densely clustered points. If any input point p in the input has at least *minpts* points within a distance of *radius*, all these points are assigned to the same cluster. The code shown here proceeds in a depth-first search, and has execution time in $\Theta(n^2)$. Efficiently converting it into a circuit, however, poses a number of challenges, which we discuss next.

The first problem we face concerns loop unrolling. Lexically, we see that the code has loops nested up to three levels deep (labelled in the figure as (A), (B) and (C)). Therefore, if we naïvely unroll it, the circuit automatically becomes $\Theta(n^3)$-sized, even though we know only $\Theta(n^2)$ iterations will actually be executed. We know this because loop (B) will be skipped if loop (A) is currently at a point that has already been assigned a cluster (this check occurs at line 6). So, we know that the body of loop (B) is executed at most only n times total, even though it is nested inside another loop. To avoid generating n^2 copies of this loop in the unrolled circuit, we simply *flatten* the loops (A) and (B) as shown in Figure 3.19. This allows us to unroll the flattened version just $2n$ times. The other loops (e.g., loop (C) or the one at line 12) were not flattened, since we do not have such strong lower bounds on how often they are skipped. Generally, such flattening tends to help only if the number of iterations taken by the inner loop can be substantially different for each iteration of the outer loop, depending on the private inputs.

However, this comes at a small additional cost: the value of the variable i can no longer be determined at

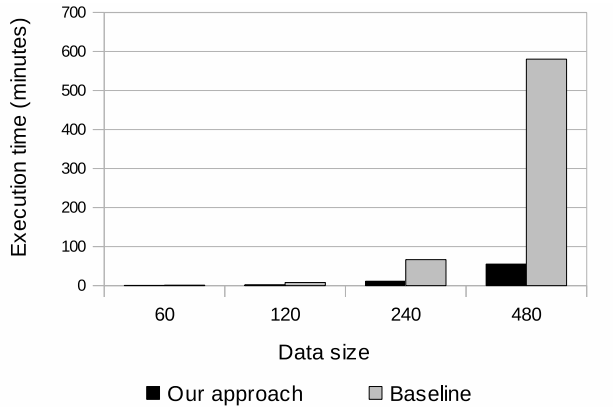


Figure 3.20: Execution time for DBSCAN clustering protocol over garbled circuits. Data size is in number of data points, where each data point is simply an (x, y) pair of two 16-bit integers, and the distance metric used is Manhattan distance.

the time of circuit generation. So, the array accesses at line 6 and line 8 will now be expensive, requiring full multiplexers. This does not occur at the innermost loop levels, and the asymptotic complexity is therefore unaffected. Most other array accesses in the program involve indices known at the time of circuit generation, and are therefore trivially implemented (e.g., unrolled version of loop (C) will have constant values for j in each copy of the loop body).

The only remaining trouble will be the stack push operation inside loop (C). Without the use of our stack construction, there is no simple way of avoiding yet another $\Theta(n)$ complexity factor here. Simply substituting a naive construction of the stack with our data structure reduces the complexity of the generated circuit from $\Theta(n^3)$ to $\Theta(n^2 \log n)$. The effect of this one simple change is shown in Figure 3.20. All other optimizations are identical in the compared versions to isolate the impact of just using our stack circuit constructions. In the case of 480 data points, the runtime drops from almost 10 hours to less than 1 hour.

3.6.3 Test Input Generation

To evaluate the impact of our structures on symbolic execution, we use the merging procedure of the merge sort algorithm. A version of this code is shown in Figure 3.21. However, the code shown has a bug: in the last loop, it uses the array elements without first checking if the index is already out of bounds. While the bug is quite simple, most popular automated tools today have a hard time detecting this bug. This is because of the well known path explosion problem, where the number of possible paths that can be taken through the code is exponential in MAXSIZE. We tried using a state-of-the-art symbolic execution tool, KLEE [18], on this example. However, it simply enumerates every single one of these paths, creating a new constraint set for each of them. As a result, it never actually generates a path that exposes the bug.

Instead, we converted the entire computation into a circuit (which is the same as fully joining all the paths together into a large constraint, as described in Section 3.3), and added constraints to let a SAT solver find an

```
1 #define MAXSIZE 100

  int merge (int *arr1, int *arr2, int n, int *dest) {
    int i, j, k;
5   if (n > MAXSIZE) return -1;
    for (i = 1; i < n; ++i)
      if (arr1[i-1] > arr1[i]) return -1;
    for (j = 1; j < n; ++j)
      if (arr2[j-1] > arr2[j]) return -1;
10  i = j = 0;
    for (k = 0; k < 2 * n; ++k) {
      if (arr1[i] < arr2[j]) dest[k] = arr1[i++];
      else dest[k] = arr2[j++];
    }
15  return 0;
  }

  int main() {
    int a[MAXSIZE], b[MAXSIZE], dest[2*MAXSIZE];
20  int n;
    fromInput (a, b, &n);
    merge (a, b, n, dest);
  }
```

Figure 3.21: A C program fragment for the merge procedure of merge sort. It has a bug since it does not check if i or j are already out of bounds in the last loop body.

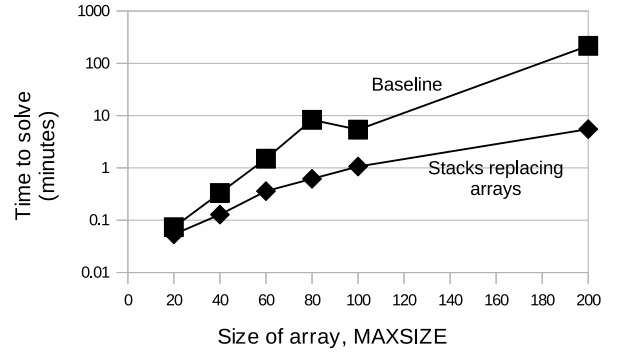


Figure 3.22: Time taken to solve for an input that triggers a buffer overflow. The scale for the y-axis is logarithmic. For the largest case, the speedup is over 30x.

input that exposes an out-of-bounds access. We solve the same problem through the SAT solver twice: in one run we change the array access on lines 12 and 13 to use our stack structure, while in the other case we leave it unchanged. The timing results are shown in Figure 3.22. The circuit structures reduce the time to find the bug from over 3 hours to just 6 minutes when the array size is 200 elements.

In practice, programs often have buffer overflow errors like these that are not triggered unless the data size is large and has a particular pattern of values. However, most complicated access patterns involving such large arrays are considered impractical for symbolic execution systems in use today. Our example here clearly shows the value in thinking of the constraint sets in terms of static circuits, and using that abstraction to create more optimized queries.

Although in our experiments we manually wrote a program that generates queries for this particular function, we expect that this process can be more automated in future. At least for the common cases described in Section 3.4, it should not be too hard for an automated tool to replace array uses with the stack and queue structures that we describe here, obtaining the same speedups we show here.

3.7 Related Work

While we include cyclic circuits in our notion of static valued circuits, the special case of acyclic circuits have been extensively studied in the past. Classical results from circuit complexity provides bounds for the sizes of many interesting families of functions. Examples of such families include symmetric functions [57], monotonic functions [5], and threshold functions [17]. Such functions tend to be too simple or restrictive for our purposes. The result most relevant to our work is Pippenger and Fischer's classical paper on oblivious Turing machines [87], where they show how any Turing machine with sequential access to a tape can be simulated in a combinatorial circuit of $\Theta(n \log n)$ size where n is the number of steps to simulate. While a sequential tape can be immediately used as a stack, it is not obvious how to extend their result to a FIFO queue, which is why we do not use their circuit. Moreover, we optimize our designs to better suit our application targets. For example, Pippenger and Fischer's design needs to support only one general operation — simulating a single Turing machine time step. This is far too general for our needs, and we generate much less expensive specialized push and pop circuits.

Besides results for circuit complexity, the two application targets that we focus on in this chapter have seen rapid improvements in speed in recent years. Below we summarize some of the important ideas that made this possible, and how they relate to our work.

3.7.1 Secure Computation — Garbled Circuits

Much of the recent work enabling fast execution of garbled circuits actually work at the level of the underlying protocol, and does not necessarily change the circuit being simulated. This includes the free XOR trick [61] that almost eliminates the cost of executing XOR gates, garbled row reductions that reduce communication overhead by 25% [83], and pipelined execution that overlaps the various phases of execution for scalability

and reduced latency [50]. Since our techniques do not depend on any of the specifics of the underlying protocol, they can be used in combination with all of these popular optimizations.

While traditionally most garbled circuit execution systems were considered practical only against an *honest-but-curious* adversary, execution of billion gate circuits against a *fully malicious* adversary has also been demonstrated recently by Kreuter et al. [63] by extensive use of parallelism. They used the cut-and-choose technique outlined by Lindell and Pinkas [68] to make their execution resistant against malicious parties. Another, more efficient, technique for fully secure execution of garbled circuits that have been proposed recently by Huang et al. is based on dual execution of the protocol [51, 75]. Since our optimization only impact the circuits, they can be used with any of these flavors of garbled circuit protocols.

Gordon et al. [45] recently demonstrated a hybrid secure computation protocol involving garbled circuits and oblivious RAM that provides general random access to arrays in sublinear time. However, their protocol still has a very high overhead due to the use of Oblivious RAM (ORAM) structures. Our purely circuit-based solutions, on the other hand, are much faster whenever they are applicable. In terms of absolute performance, the fastest they reported was about 9.5 seconds per element access, and that for an array of 2^{18} elements. Because of such high overheads, it was actually still faster to naively multiplex over the entire array unless the array is big (in their implementation, they break-even at arrays of approximately 260,000 elements). Our circuit-structures are orders of magnitude faster in the cases we can handle, and as shown in Section 3.6 we breakeven for much smaller data sizes.

Finally, many of the recent frameworks and compilers such as Fairplay [73] and CMBC-GC [48] provide ways to produce garbled circuit protocols starting from high-level programs. Although we have not focused on automatic circuit compilation in this work, we hope that in future such tools will be able to automatically detect which circuit structure is applicable a given situation. This would allow programmers to write code naturally using standard data structures like arrays, but generate circuits that automatically implement array accesses using appropriate less expensive data structures to achieve reasonable performance.

3.7.2 Symbolic Execution

The development of symbolic execution as a tool for static analysis has, in large part, been aided by the concurrent improvements in constraint solvers. With modern constraint solvers, symbolic execution systems such as KLEE [18] and EXE [19] are able analyze a program and solve for inputs that drive execution of the given program along a certain path. We hope that our methods would make it easier for such tools to handle far more complicated programs than they are currently able to.

More recent advances in this field mostly focused on improving scalability of these tools and on solving the exponential path explosion problem. Path explosion is a notorious problem where the number of paths to be explored is exponential in the number of branches along that path. Solutions recently proposed include compositional methods and state merging. Compositional approaches (e.g., [4, 36, 38]) attempt to keep the paths shorter by analyzing one function at a time and composing the results together later, often lazily. In practice, however, state joining [46, 65] seems to provide better results where several paths are merged into

one larger query the way we did here. Some researchers have also noted how this method can be seen as a strict generalization of the compositional methods [65].

Something we did not delve into in this paper is how to determine loop bounds. Since current systems require loops to be completely unrolled, tools need to establish a reasonable upper bound for how many iterations of the loop should be explored. There has been a lot of recent work in this area, all of which complements our work on arrays. For example, Obdržálek and Trtík [81] recently showed how integer recurrence equations can sometimes be used to solve for an upper bound. Saxena et al. demonstrated how using input grammar specification can sometimes help determine loop bounds [91]. Given how loops are often used in conjunction with arrays, we believe our work will further broaden the scope of their techniques to more complicated coding patterns.

3.8 Conclusion

We showed how a common set of ideas can be used to speed up common programming patterns in generic secure computation and symbolic execution of programs, two previously unrelated applications. Both of these applications depend on static circuits. We devised circuit structures that lead to large speedups for several common data structures. Further, we demonstrated how slow array access operations in programs can often be replaced by more specialized data structures like stack, queue, and associative map, achieving asymptotic improvements in runtime. We are optimistic that similar approaches can be taken with other data structures to provide similar gains for a wide range of applications. Although our work focused on manual implementation, we believe such transformations could be automated in many cases, and our results point to future opportunities for automatically generating efficient static circuits for secure computation and symbolic execution.

Chapter 4

Oblivious RAM[8]

4.1 Introduction

So far, in Chapter 3, we have only seen techniques for efficiently performing memory accesses when there is a specific, known access pattern that we can take advantage of. For general random access, however, we need to make use of Oblivious RAM (ORAM). The naïve approach for hiding access location would be to linearly scan the entire array. However, in native computation, such accesses take constant-time. To fill this gap, researchers have investigated secure computation in the *random access machine* (RAM) model of computation [26, 34, 45, 56, 70–72, 103, 108]. These suggest various methods for constructing ORAMs that provide sub-linear access time while hiding access patterns.

ORAM protocols were originally proposed for a client-server setting where a client stores and manipulates an array of length n on an untrusted server without revealing the data or access patterns to the server. Gordon et al. adapted ORAM to the setting of secure computation [45], where parties collectively maintain a memory abstraction that they can jointly access access, while hiding the access patterns from everyone. In essence, the parties run a secure-computation protocol to store *shares* of the state of the underlying ORAM protocol, and then use circuit-based secure computation to execute the ORAM algorithms.

Although there is a rich literature devoted to developing ORAM protocols with improved performance [13, 43, 44, 64, 85, 95, 96, 104], most of this literature focuses on optimizing performance in the client-server setting, and most work on RAM-based secure computation (RAM-SC) uses existing ORAM protocols (to a first approximation) as black boxes. We highlight, however, that there are a number of differences between applications of ORAM in the two settings:

1. In the client-server setting the client owns the data and performs the accesses, so the privacy requirement is unilateral. In the RAM-SC setting none of the parties should be able to learn anything about

[8]This chapter is an adaptation of:
Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, Jonathan Katz. Revisiting Square-Root ORAM Efficient Random Access in Multi-Party Computation. In 37th IEEE Symposium on Security and Privacy, 2016, San Jose.

the data or access patterns.

2. In the client-server setting the client's state should be sublinear in n or else the problem is trivial; for RAM-SC, however, the linear state is stored across both parties.

3. In the client-server setting the most important metric is the total communication complexity. In the RAM-SC setting other measures of efficiency become more important. Specifically, the algorithmic complexity is important because the algorithms will be emulated using generic secure computation.

4. In the client-server setting, the initialization step (when the client outsources its data to the server) is "free" because it is a local action on the part of the client. In the RAM-SC case, the parties must use a distributed protocol for initialization and the cost of doing so may be prohibitive.

Existing work on ORAM has focused entirely on *asymptotic* performance; we are not aware of any prior work whose aim is to improve performance for concrete values of n. Indeed, prior work in the RAM-SC setting has found that a linear scan over the data (i.e., a trivial ORAM construction) outperforms more-complicated ORAM constructions until n becomes quite large [45, 101, 102] (in practice, n is often small even when the inputs are large since n may denote the length of a single array rather than the entire memory being used by the computation, and each memory block may contain many individual data items). This means that for practical sizes, the entire body of research on ORAM has had little impact as far as RAM-SC is concerned.

Contributions. We re-visit the classical square-root ORAM of Goldreich and Ostrovsky [40], and propose a number of modifications to that construction with the goal of obtaining an ORAM scheme suitable for secure computation in the semi-honest setting:

1. In the original scheme, the client uses a hash function to compute the *position map* (i.e., the mapping from semantic addresses to physical addresses). We replace this with a shared array storing the position map explicitly. This is particularly beneficial when the underlying ORAM algorithms are computed using generic circuit-based secure computation.

2. Because the position map is stored explicitly, initialization and reshuffling (expensive operations performed sporadically) can be made much more efficient than in the original construction, as they can be based upon Waksman shuffling networks [100] rather than oblivious sorting.

3. As observed in prior work [96] the position map is a constant factor smaller than the original memory array. We use ORAMs recursively to enable oblivious access to the position map, and develop a number of optimizations in order to obtain a secure and efficient protocol.

We implement and evaluate our construction (code available at `http://oblivc.org/`) and show that for small-to-moderate values of n our scheme offers more efficient data access than Circuit ORAM [101]. In fact, our scheme outperforms even the trivial ORAM (i.e., linear scan) for n as small as 8 (the exact crossover

point depends on the block size used as well as the underlying network and processor). Our construction also significantly outperforms prior work in terms of initialization time. To understand how the properties of different applications impact ORAM performance, and demonstrate the general applicability of our design, we implement and evaluate several benchmark application, including secure two-party computations of the Gale-Shapley stable matching algorithm, breadth-first search, binary search, and the Scrypt hash function. The resulting protocols are more efficient than prior approaches by an order of magnitude or more in some cases.

4.2 Background

This section provides a brief introduction to oblivious RAM (ORAM) and RAM-based secure computation (RAM-SC), and closely related protocols.

4.2.1 Oblivious RAM

Oblivious RAM provides a memory structure that hides access patterns [40]. An ORAM scheme consists of two protocols: an *initialization protocol* that takes as input an array of elements, initializes a new oblivious structure in memory; and an *access protocol* that implements each logical access to the ORAM with a sequence of physical accesses to the underlying structure.

To be secure, an ORAM must satisfy two properties: 1) the physical access pattern of the initialization protocol is indistinguishable when initializing different input arrays of the same size; 2) for any two sequences of semantic accesses of the same length, the physical access patterns produced by the access protocol must be indistinguishable. Note that it is always possible to implement a secure initialization protocol by performing the access protocol iteratively on all input elements, and this is the approach taken by previous ORAM designs used in RAM-SC. It can be, however, very inefficient to initialize the ORAM through repeated accesses.

Goldreich and Ostrovsky [40] introduced two ORAM constructions with a hierarchical layered structure: the first, *Square-Root ORAM*, provides square root access complexity; the second, *Hierarchical ORAM*, requires a logarithmic number of layers and has polylogarithmic access complexity. A recent series of ORAM schemes, beginning with the work of Shi et al. [95], adopted a sequence of binary trees as the underlying structure. While, asymptotically, the most bandwidth efficient ORAM constructions known use the hierarchical paradigm [64], tree-based ORAMs are considered more efficient for practical implementations especially when used in MPC protocols. This is primarily because classical hierarchical constructions use hash functions or pseudorandom functions (PRFs) to shuffle data in the oblivious memory. In an MPC context these functions must be executed as secure computations with large circuits.

4.2.2 RAM-Based Secure Computation

In traditional MPC, general input-dependent array access incurs a linear-time overhead since all elements need to be touched to hide the position of interest. RAM-based secure computation (RAM-SC) combines ORAMs with circuit-based MPC protocols, to enable secure random memory accesses [45]. In RAM-SC, the bulk of the computation is still performed by a circuit-based protocol as in traditional MPC, but memory accesses are performed using an ORAM that is implemented within the MPC protocol. For each access, the circuit now emulates an ORAM access step to translate a secret logical location into multiple physical locations that must be accessed. The physical locations are then revealed to the two parties, which pass the requested elements back into the circuit for use in the oblivious computation. Finally, the circuit produces new data elements to be written back to those physical positions, hiding which elements were modified and how they were permuted. One such structure is maintained for each array that needs input-dependent general random access.

Two-party RAM-SC was first implemented by Gordon et al. [45] with based on a tree-based ORAM scheme proposed by Shi et al. [95]. Subsequent works [26, 34, 56, 103] presented improved protocols, all based on tree-based ORAM constructions. Wang et al. [101] proposed Circuit ORAM, which yields the best known circuit size both in terms of asymptotic behavior and concrete performance. In Section 4.5, we provide performance comparisons between our new ORAM scheme and Circuit ORAM, showing orders of magnitude improvement for access and initialization across a wide range of parameters and applications.

4.2.3 Variations

In addition to the RAM-SC model we focus on, there are other uses for ORAMs in secure computation protocols. Some of the ORAM innovations produced in these settings have been applied to the RAM-SC designs in Section 4.2.2. Although it is beyond the scope of this work, we believe our ORAM design may likewise yield benefits in other contexts.

Gentry et al. [34] proposed several optimizations for tree-based ORAMs and considered briefly how to build a HE-over-ORAM system. A system based on Path ORAM [96] was built in their subsequent work [35]. They showed a per-access time of 30 minutes for a database with 4 million 120-bit records, excluding the cost of initialization.

Lu and Ostrovsky [72] designed an ORAM algorithm based on two non-colluding servers. When applied to a two-party secure RAM computation setting, these servers become parties engaging in an MPC protocol. Their construction achieves $O(\log N)$ overhead, but suffers from huge concrete costs because it requires oblivious evaluation of $\Theta(\log N)$ cryptographic operations per access, which is prohibitively expensive in an MPC protocol.

Afshar et al. [3] discussed how to extend RAM-SC with malicious security, where both parties can arbitrarily deviate from the protocol. They proposed efficient consistency checks that avoid evaluating MAC in circuits. In this chapter, we only consider semi-honest adversaries, and hope that future work will extend our

protocol to be secure against malicious adversaries.

4.3 Revisiting Square-Root ORAM

In this section we revisit Goldreich and Ostrovsky's square-root ORAM design [40] and adapt it to the RAM-SC setting. Section 4.3.1 introduces notations used to describe ORAM algorithms; Section 4.3.2 provides a brief description of the original scheme; Section 4.3.3 introduces a basic (but inefficient) construction by making some key changes to the original scheme; Section 4.3.4 shows how to improve its efficiency with a recursive construction which is our final design.

4.3.1 Notation

We use $\langle x \rangle$ to denote a variable x secretly shared by the two parties. In our garbled circuit implementation, $\langle x \rangle$ means the generator knows (k^0, k^1) and the evaluator knows k^x. Since the actual value of x is not known to either party, we interchangeably use the terms "private", "garbled", and "oblivious" to describe it.

The length of an array is always public, although padding can be used to hide its exact length when necessary. An array containing private elements is denoted using angle brackets (e.g., $\langle \mathsf{Array} \rangle$). We denote the i^{th} element of an array using a subscript (e.g., $\langle \mathsf{Array} \rangle_i$). The index may be oblivious (e.g., $\langle \mathsf{Array} \rangle_{\langle i \rangle}$), in which case the array access is performed via linear scan.

The structure blocks represents an array of block objects. Each block contains private data, $\mathsf{block}.\langle \mathsf{data} \rangle$, and a private record of its logical index, $\mathsf{block}.\langle \mathsf{index} \rangle$. Thus, i is the physical index of blocks_i, and $\mathsf{blocks}_i.\langle \mathsf{index} \rangle$ is the logical index of the same block. Neither changing the contents of a block nor moving it from one structure to another alters its logical index, unless explicitly noted.

In pseudocode, ordinary conditional statements will use the keyword if, while conditionals on secret values will use $\langle$**if**$\rangle$. The bodies of secret conditionals are always executed, but the statements in them are executed conditionally, becoming no-ops if the condition is false.

We use $\langle a \rangle \xleftarrow{\$} B$ to denote random choice of a secret element a from a public set B.

Figure 4.1 shows how the access algorithm for a naïve linear scan ORAM is written in our notation. The algorithm Access takes three parameters as inputs:

- Oram: the main data structure storing the payload.
- $\langle i \rangle$: the private, logical index of the block we want to access.

```
define Access(Oram, ⟨i⟩, Φ):
  for j from 0 to Oram.n − 1:
    ⟨if⟩ ⟨i⟩ = j: Φ(Oram_j)
```

Figure 4.1: Access algorithm for the linear scan ORAM.

```
define Write(Oram, ⟨i⟩, ⟨val⟩):
  define Φ(block):
    block.⟨data⟩ ← ⟨val⟩
  Access(Oram, ⟨i⟩, Φ)

define Read(Oram, ⟨i⟩):
  ⟨val⟩ ← ⊥
  define Φ(block):
    ⟨val⟩ ← block.⟨data⟩
  Access(Oram, ⟨i⟩, Φ)
  return ⟨val⟩
```

Figure 4.2: Read and write wrappers defined using Access()

- $\Phi()$: a function that is invoked during access to read, write or modify the desired block.

The ORAM hides index $\langle i \rangle$ by performing a linear scan over all elements. Note that we use $\langle \textbf{if} \rangle$ for the conditional, so the body of the conditional statement will actually be executed n times, although only one will take effect. Both parties will see the garbled keys representing $\langle \mathsf{val} \rangle$ or $\langle \mathsf{data} \rangle$ change n times inside $\Phi()$; they just won't know if the associated plaintext has also changed, since that depends on secret index $\langle i \rangle$.

Users will not typically use ORAMs by directly invoking Access, but by using the wrapper functions shown in Figure 4.2. These wrappers are exactly the same across all ORAM constructions we consider; the essential logic is in Access.

4.3.2 Square-Root ORAM

Figure 4.3a shows the original square-root ORAM proposed by Goldreich and Ostrovsky [40]. The ORAM structure consists of following components:

1. Shuffle: an array of blocks, also referred to as "permuted memory" in the original paper.
2. Stash: an array of blocks, termed the "shelter" in the original paper.
3. π: a pseudorandom function (PRF) mapping indices to random strings. Note that π needs to be evaluated securely using MPC protocols, which is why previous RAM-SC designs dismissed the square-root ORAM construction.

To initialize an ORAM from an array of blocks, we first append $\sqrt{n}$ dummy blocks to the input array and obliviously permute all $n+\sqrt{n}$ blocks according to the pseudorandom permutation $\pi(i)$. Once the blocks are shuffled, their physical locations and semantic indices are uncorrelated, and so each block may be accessed once (and only once) without revealing anything about the access pattern. But, if a location in the shuffled array is accessed multiple times that would leak information, revealing that the access sequence contains a repeated access.

```
define Initialize(blocks, T)
  n ← |blocks|
  ⟨π⟩ ← pseudorandom function
  append √n dummy blocks to Shuffle
  Shuffle ← ObliviousSort(blocks, ⟨π⟩)
  Oram ← (n, t ← 0, T, ⟨π⟩, Shuffle, Stash ← ∅)
  return Oram

define Access(Oram, ⟨i⟩, Φ)
  ⟨found⟩ ← false
  for j from 0 to Oram.t:
    ⟨if⟩ Oram.Stash_j.⟨index⟩ = ⟨i⟩:
      ⟨found⟩ ← true
      Φ(Oram.Stash_j)
  ⟨if⟩⟨found⟩ :
    ⟨k⟩ ← Oram.n + Oram.t
  ⟨else⟩ : ⟨k⟩ ← ⟨i⟩
  p ← reveal(π(⟨k⟩))
  ⟨if⟩ not ⟨found⟩ :
    Φ(Oram.Shuffle_p)
  append Oram.Shuffle_p to Oram.Stash
  Oram.Shuffle_p ← dummy
  Oram.t ← Oram.t + 1
  if Oram.t = Oram.T:
    blocks ← real blocks in
      Oram.Shuffle ∪ Oram.Stash
    Oram ← Initialize(blocks, Oram.T)
```

(a) The original square-root ORAM scheme [40]

```
define Initialize(blocks, T)
  n ← |blocks|
  ⟨π⟩ ← random permutation on n elements
  Shuffle ← ObliviousPermute(blocks, ⟨π⟩)
  Oram ← (n, t ← 0, T, ⟨π⟩, Shuffle,
          Used ← ∅ , Stash ← ∅)
  return Oram

define Access(Oram, ⟨i⟩, Φ)
  ⟨found⟩ ← false
  for j from 0 to Oram.t:
    ⟨if⟩ Oram.Stash_j.⟨index⟩ = ⟨i⟩:
      ⟨found⟩ ← true
      Φ(Oram.Stash_j)
  ⟨if⟩⟨found⟩ :
    ⟨p⟩ ←$ {0, ..., (Oram.n − 1)} \ Oram.Used
  ⟨else⟩ : ⟨p⟩ ← Oram.⟨π⟩_⟨i⟩
  p ← reveal(⟨p⟩)
  ⟨if⟩ not ⟨found⟩ :
    Φ(Oram.Shuffle_p)
  append Oram.Shuffle_p to Oram.Stash
  Oram.Used ← Oram.Used ∪ {p}
  Oram.t ← Oram.t + 1
  if Oram.t = Oram.T:
    for j from 0 to |Oram.Used| − 1:
      p' ← Oram.Used_j
      Oram.Shuffle_p' ← Oram.Stash_j
    Oram ← Initialize(Oram.Shuffle, Oram.T)
```

(b) Our basic square-root ORAM scheme.

Figure 4.3

To access logical index $\langle i \rangle$, we first linear scan the blocks in Stash. If the block is not found, we compute its physical location, $\langle p \rangle \leftarrow \pi(\langle i \rangle)$; otherwise, we find the physical location of the next unaccessed dummy blocks, $\langle p \rangle \leftarrow \pi(n + \mathsf{Oram}.t)$. The value $\langle p \rangle$ is then revealed to both parties, but leaks no information about the logical index. The block at the physical location referred to by p is accessed, by doing a binary search over the $\pi(i)$ values, and moved to Stash.

After T accesses have been performed, we obliviously remove all dummy blocks in Oram.Stash and Oram.Shuffle and re-initialize the whole structure. The complete protocol is shown in pseudocode in Figure 4.3a. In Goldreich and Ostrovsky's original protocol, Oram.T is set to $\sqrt{n}$, resulting in $O(\sqrt{n}\log_2 n)$ amortized cost per access.

The original scheme was not designed for a RAM-SC setting, and suffers from two key problems that make it very expensive to implement in an MPC:

1. It evaluates the PRF $\pi(x)$ for each access; in the initialization algorithm, $n + \sqrt{n}$ evaluations of $\pi(i)$ are needed. This is inefficient, especially in MPC protocols since evaluating each PRF requires tens of thousands of gates.

2. It requires a $\Theta(n \log^2 n)$ oblivious sort on the data blocks in two different places: to shuffle data blocks according to the PRF results, and to remove dummy blocks before initialization.

Next, we discuss how to adapt the scheme for efficient use in RAM-SC by eliminating these problems.

4.3.3 Basic Construction

Figure 4.3b presents our basic construction, a step towards our final scheme. The construction is similar to the original scheme, with a key difference: instead of using PRF to generate a random permutation, it stores the permutation π explicitly as a private array. This enables several performance improvements:

1. Storing the permutation π as a private array enables us to replace oblivious sorting during the initialization with a faster oblivious permutation. In addition, the value p revealed during the access refers to the real location, which avoids using binary search to find the location for p. Section 4.3.4 shows how to recursively implement π for better efficiency.

2. We eliminate the need of dummy blocks. When a dummy access is needed, we instead access a random location for real blocks that is not accessed before and append the block to the Stash.

3. By using a public set Used, we avoid the oblivious sorting needed when moving blocks from the Stash to Shuffle. This is efficient since Used is maintained in the clear and is secure because all elements in Used have already been revealed.

Security. Assuming the MPC protocol itself is secure and does not leak any information about oblivious variables, this protocol satisfies the ORAM requirement that no information is revealed about the logical

access pattern. On each access, a uniform unused element from Shuffle is selected, regardless of the semantic index requested. Subsequently, the entire Stash is always scanned. Finally, the entire structure is reshuffled at a fixed interval, in a manner independent of the access pattern. The only values revealed are the permuted physical indices p; the set Used, which contains no information about the semantic indices; and the counter t, which increments deterministically.

Asymptotic cost. Now we analyze the average cost of accessing a block in this basic scheme. We represent the combined cost of accessing $\langle\pi\rangle$ and Used as $c(n)$, some value that only depends on the number of blocks, n, but not block size. We use B to denote the cost of accessing a single block (this could be bandwidth, time, or energy cost). The augmented cost, $B' = B + \Theta(\log_2 n)$, includes the additional cost of accessing the metadata containing the block's logical index. For an ORAM of size n, each logical index requires $\log_2 n$ bits, so it incurs $\Theta(\log_2 n)$ cost to retrieve or compare an index.

Since our construction is a periodic algorithm that performs a shuffle every T accesses, we obtain the amortized per-access cost by computing the average over T accesses. This is the cost of the shuffle plus the cost of B' for each block touched thereafter until the next shuffle.

The cost of shuffling is approximately $B'W(n)$ using a Waksman network [100]. Here, $W(n) = n\log_2 n - n + 1$ is the number of oblivious swaps required to permute n elements. On each access, the entire Stash, comprising t blocks, must be scanned. Thus, the total cost of the T accesses and one shuffle which constitutes a full cycle is given by

$$\begin{aligned} & B'W(n) + \sum_{t=1}^{T} \left(B't + c(n)\right) \\ \leq\ & B'n\log_2 n + \frac{1}{2}B'T(T+1) + Tc(n) \\ =\ & T\left(\frac{1}{T}B'n\log_2 n + \frac{1}{2}B'(T+1) + c(n)\right) \\ =\ & TF(n) \end{aligned}$$

where $F(n)$ is the amortized per-access cost we are after.

If reshuffle period $T = \Theta(\sqrt{n\log_2 n})$, the asymptotic cost is $F(n) = \Theta(B'\sqrt{n\log_2 n})$, assuming the block size is large enough to make $c(n)$ negligible compared to B.

Concrete cost. This design is less expensive than linear scan, even for reasonably small block sizes and for block counts as low as four. With linear scan, the cost is nB per access, ignoring smaller terms that are independent of B. With four blocks, the cost of a linear scan is $4B$. Using a shuffling period of $T = 3$, we get a cost of $B(W(4) + 1 + 2 + 3) = 11B$ for three accesses, again ignoring smaller terms that are independent of B. This is slightly better than the linear scan cost for three accesses, $3 \times 4B = 12B$. Thus, for four blocks of a large enough size, the simplified one-level square-root ORAM is less expensive than a linear scan, even after accounting for the cost of initialization. However, in the case of small blocks, the terms independent of B (which we have ignored) become significant enough that linear scan has a slight advantage.

```
define Initialize(blocks)
  n ← |blocks|
  ⟨π⟩ ← random permutation on n elements
  Shuffle ← ObliviousPermute(blocks, ⟨π⟩)
  T ← ⌈√W(n)⌉
  Oram_1 ← InitializePosMap(⟨π⟩, 1, T)
  Oram_0 ← (n, t ← 0, T, Oram_1, Shuffle,
            Used ← ∅, Stash ← ∅)
  return Oram_0

define Access(Oram_0, ⟨i⟩, Φ, )
  ⟨found⟩ ← false
  for j from 0 to Oram.t:
    ⟨if⟩ Oram_0.Stash_j.⟨index⟩ = ⟨i⟩:
      ⟨found⟩ ← true
      Φ(Oram_0.Stash_j)
  p ← GetPos(Oram_0.Oram_1, ⟨i⟩, ⟨found⟩)
  ⟨if⟩ not ⟨found⟩:
    Φ(Oram_0.Shuffle_p)
  append Oram_0.Shuffle_p to Oram_0.Stash
  Oram_0.Used ← Oram_0.Used ∪ {p}
  Oram_0.t ← Oram_0.t + 1
  if Oram_0.t = Oram_0.T:
    for j from 0 to Oram_0.T − 1:
      p′ ← Oram.Used_j
      Oram_0.Shuffle_p′ ← Oram_0.Stash_j
    Oram_0 ← Initialize(Oram_0.Shuffle)
```

Figure 4.4: Our recursive square-root ORAM scheme. $W(n)$ is the number of swaps needed in a n-sized Waksman permutation network.

In our experiments, we observed the square-root scheme to be more efficient in terms of bandwidth for four blocks of just 36 bytes each (see Section 4.5.2 for details). For larger block sizes, we found that the cost ratio reaches 11 : 12, as expected.

4.3.4 Scalable Construction

So far, we have not discussed how to implement the structure $\langle\pi\rangle$ more efficiently than linear scan, aside from claiming that its costs do not depend on the block size. For small values of n, linear scan is good enough, as in the four-block example above. At this size, π comprises just four records of two secret bits each. However, for larger values of n, it may seem natural to build these structures upon recursive ORAMs of decreasing size. As we discuss next, however, this method is unacceptably costly. Our solution is to specialize the structure for position maps.

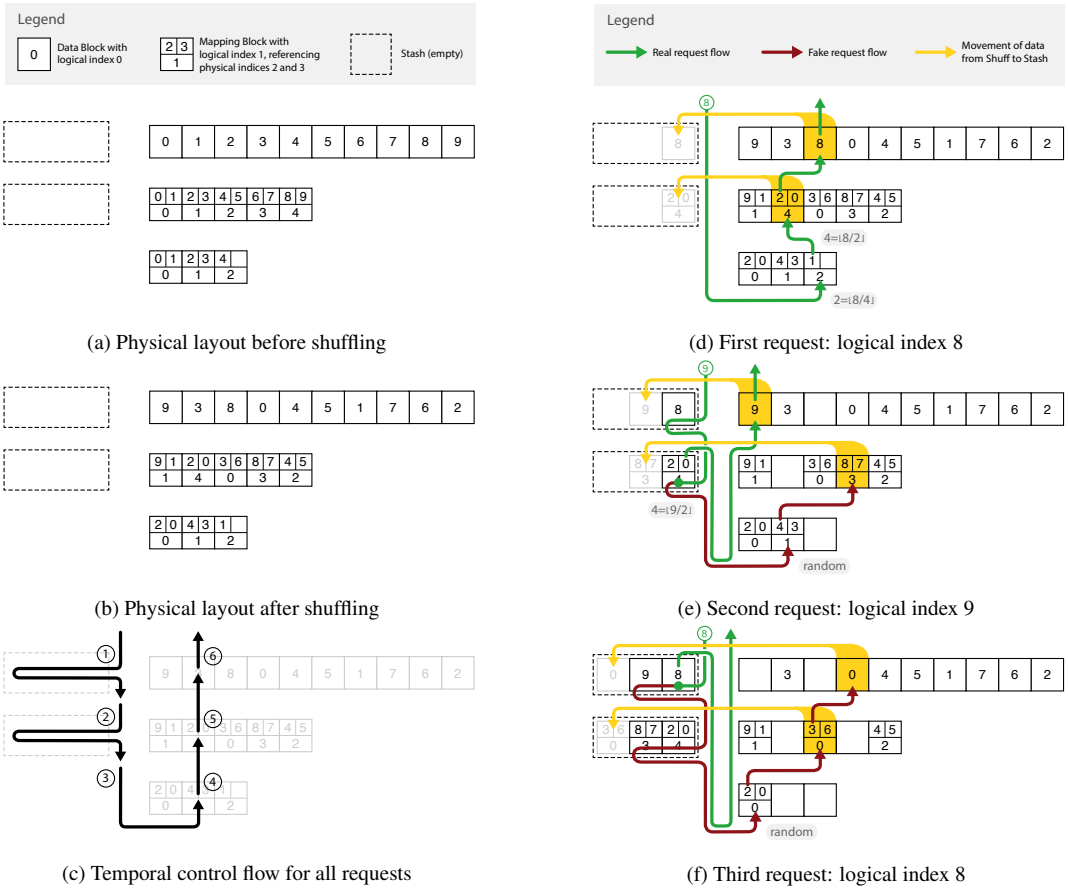
The position map structure, $\langle\pi\rangle$, is common to most existing tree-based constructions [95, 96, 101]. It is usually implemented atop recursive ORAMs of decreasing size, each level packing multiple indices of the previous into a single block, and the whole thing is updated incrementally as elements of the main ORAM are accessed. In these constructions, each ORAM lookup requires a single corresponding lookup in each recursive position maps. However, in our scheme, a naïve recursive structure for $\langle\pi\rangle$ would require $n+T$ position lookups for every T accesses to the main ORAM (where T is the number of accesses between shuffles) since each of the T main accesses would require an access to the position map, and additional n accesses would be required to store the regenerated permutation π' when the ORAM is shuffled.

```
define InitializePosMap(⟨π⟩, k, T)
  n ← |⟨π⟩|
  if n/pack ≤ T:
    ⟨Used_{0···(n−1)}⟩ ← (false, ..., false)
    Oram_k ← (n, t ← 0, T, ⟨π⟩, ⟨Used⟩)
  else:
    for i ∈ {0 ... ⌈n/pack⌉ − 1}:
      ⟨data⟩ ← (⟨π⟩_{pack·i}, ..., ⟨π⟩_{pack·(i+1)−1})
      blocks_i ← (⟨data⟩, ⟨index⟩ ← i)
    ⟨π′⟩ ← random permutation on ⌈n/pack⌉ elements
    Shuffle ← ObliviousPermute(blocks, ⟨π′⟩)
    Oram_{k+1} ← InitializePosMap(⟨π′⟩, k+1, T)
    Oram_k ← (n, t ← 0, T, Oram_{k+1}, Shuffle,
              Stash ← ∅)
  return Oram_k

define GetPosBase(Oram_k, ⟨i⟩, ⟨fake⟩)
  ⟨p⟩ ← ⊥
  ⟨done⟩ ← false
  for j from 0 to (Oram_k.n − 1):
    ⟨s_1⟩ ← (not ⟨fake⟩ and ⟨i⟩ = j)
    ⟨s_2⟩ ← (⟨fake⟩ and not Oram_k.⟨Used⟩_j
                and not ⟨done⟩)
    ⟨if⟩ ⟨s_1⟩ or ⟨s_2⟩:
      ⟨p⟩ ← ⟨π_j⟩
      Oram_k.⟨Used⟩_j ← true
      ⟨done⟩ ← true
  p ← reveal(⟨p⟩)
  return p

define GetPos(Oram_k, ⟨i⟩, ⟨fake⟩)
  if Oram_k.n/pack ≤ Oram_k.T:
    p ← GetPosBase(Oram_k, ⟨i⟩, ⟨fake⟩)
  else:
    ⟨found⟩ ← false
    ⟨h⟩ ← ⟨i⟩/pack
    ⟨l⟩ ← (⟨i⟩ mod pack)
    for j from 0 to Oram_k.t − 1:
      ⟨if⟩ Oram_k.Stash_j.⟨index⟩ = ⟨h⟩:
        ⟨found⟩ ← true
        block ← Oram_k.Stash_j
        ⟨p⟩ ← block.⟨data⟩_{⟨l⟩}
    p′ ← GetPos(Oram_{k+1}, ⟨h⟩, ⟨fake⟩ or ⟨found⟩)
    append Oram_k.Shuffle_{p′} to Oram_k.Stash
    Oram_k.t ← Oram_k.t + 1
    ⟨if⟩ ⟨fake⟩ or not ⟨found⟩:
      block ← Oram_k.Stash_{(Oram_k.t−1)}
      ⟨p⟩ ← block.⟨data⟩_{⟨l⟩}
    p ← reveal(⟨p⟩)
  return p
```

Figure 4.5: Implementation of the recursive position map.

This is a serious problem: each level of the recursive structure would need to store pack indices of the previous level in a single block, which would be traversed by linear scan. Thus, each subsequent level decreases in element count by a factor of pack, but all levels require pack time to linear scan the relevant block. We can multiply by $(n+T)/T$ to amortize the cost over T accesses, where $T = \sqrt{n \log_2 n}$, the shuffle period (as computed in Section 4.3.3). If the amortized cost per access to level i of this map is $c_i(n)$, we have:

$$\begin{aligned} c_i(n) &\geq \frac{n+T}{T}(c_{i+1}(n/\mathsf{pack}) + \mathsf{pack}) \\ &\geq \frac{n}{\sqrt{n \log_2 n}} c_{i+1}(n/\mathsf{pack}) \\ &\geq \sqrt{\frac{n}{\log_2 n}} c_{i+1}(n/\mathsf{pack}). \end{aligned}$$



(a) Physical layout before shuffling

(b) Physical layout after shuffling

(c) Temporal control flow for all requests

(d) First request: logical index 8

(e) Second request: logical index 9

(f) Third request: logical index 8

Figure 4.6: Illustration of data flow for one full cycle of an example ORAM. In subfigures (d), (e), and (f) we present the logical dependencies for three sequential accesses.

This is a super-polynomial function with $\Theta(\log n)$ levels of recursion, which is unacceptable for our efficiency goals. Fixing this involves three changes to our basic construction.

The first change is to take advantage of our ability to initialize quickly from an oblivious array. On each shuffle, we regenerate π, and, instead of writing it into the recursive structure element by element, we re-initialize the recursive structure using π' as the seed data. This eliminates the extra n accesses to the position map on each cycle.

Second, we lock all levels of the recursive structure to the same shuffle period, $T = \sqrt{n \log_2 n}$, where n is the number of blocks in the main ORAM (the level that contains the original data). We terminate the recursion at the first level with fewer than T blocks, and access this final level using linear scan. Using this arrangement, we can initialize the entire ORAM in $\Theta(Bn \log n)$ bandwidth and time.

This second modification has a downside. All levels of the recursive ORAM shuffle in synchronization

with one another, based on a shuffle period determined by the largest level. This shuffle period will be significantly suboptimal for levels with fewer blocks. We pay a time and bandwidth cost of $\Theta(T)$ at each level of the ORAM (for linearly scanning the $\Theta(T)$ blocks in each level's Stash). An ORAM instantiated with n elements will have $\log n$ levels, which brings the cost per access to $\Theta(T \log n) = \Theta(\sqrt{n \log^3 n})$. However, the linear scan overhead incurred by using a global shuffling period is compensated for by gains in the efficiency of Used which it enables.

Constructing an efficient mechanism for keeping track of used and unused physical blocks poses a challenge. Used contains inherently public data — both parties are aware which physical locations in Shuffle have already been accessed — yet, they must obliviously check whether it contains a secret logical index $\langle p \rangle$. Moreover, they must be able to sample a secret, uniform element from $S = \{0, \ldots, n-1\} \setminus \mathsf{Used}$. The simplest method would be to sample an integer from $\{0, \ldots, |S|-1\}$ and then obliviously map it to the set S, an expensive operation.

The third change removes the need to obliviously check Used for secret index $\langle p \rangle$. Instead of using an explicit data structure, our choice of a global shuffle period allows us to implicitly represent a superset of Used in the recursive structure $\langle \pi \rangle$, by tracking which blocks the smallest recursive level have been used. We use the notation Oram_k.Stash, Oram_k.Shuffle, and Oram_k.Used to represent the corresponding structures in recursive ORAM at level k. Oram_0 is the main ORAM that holds the data blocks; Oram_1 is the top level of the position map $\langle \pi \rangle$; Oram_2 and so on indicate deeper levels of the recursive position map structure.

We maintain the invariant that if a block has already been moved from Oram_n.Shuffle to Oram_n.Stash, the corresponding block in Oram_{n+1} has also been moved from Oram_{n+1}.Shuffle to Oram_{n+1}.Stash. The converse is not necessarily true: it is possible for Oram_{n+1}.Stash to contain blocks that map to unaccessed blocks in Oram_n. This can happen, for example, if logical block i of Oram_0 has been accessed and block $i+1$ has not, but mapping information for both blocks resides in the same block of Oram_1.

Randomly sampling an unused block with this construction is simple. At the smallest level the blocks are linearly scanned, so we just pick the first unused element. This is guaranteed to point to a random unused position. At the next recursive level, we can use any element in the block referred to by the index from the first level, since they are all random and unused. The process continues to ripple upward until an unused block in the required ORAM level has been selected. This method excludes from the set to be randomly sampled any block referred to by a block that has been accessed at a lower level. Nonetheless, blocks sampled randomly remain indistinguishable from genuine accesses, as, for each top level access, exactly one unused block is accessed at each lower level.

The final construction is presented in Figures 4.4 and 4.5, and the life-cycle of the ORAM is illustrated in Figure 4.6.

4.4 Techniques and Optimizations

This section presents some of the lower-level techniques used in our implementation.

Shuffling. We employ a Waksman network [100] for shuffling. The network executes many oblivious swap operations, each controlled by a secret bit determined by the permutation π. Let B be the number of bytes transferred when obliviously swapping two blocks of data. Since a Waksman network for shuffling requires $W(n) = n\log_2 n - n + 1$ swap operations, it is expected that the two parties will transfer $BW(n)$ bytes during a shuffle, excluding the secret control bits.

The control bits pose a problem: neither party can learn anything about the randomly sampled permutation π, but we do not know an efficient oblivious algorithm for computing the corresponding control bits. To solve this problem, we perform two shuffles: the parties locally a pick secret permutation each and compute their corresponding control bits in the clear. Each party's local permutation constitutes its share in the final secret permutation π, which is the composition of the two permutations. So long as at least one party behaves honestly, the result is a uniformly random permutation, discoverable to neither. They can jointly shuffle the data by running two consecutive shuffling networks, one for each permutation.

Performing a shuffle in this way is quite inexpensive. The bandwidth cost of $2W(n)$ swaps is comparable to $W(n)$ AND gates, using the oblivious shuffle design from Huang et al. [49] and half-gates technique from Zahur et al. [109]. However, each time we perform a shuffle, we incur the latency of a network round-trip, since the evaluator retrieves new garbled labels for control bits via oblivious transfer extension [7].

Computing the permutation. Whenever the data in Shuffle is shuffled, we must reinitialize the recursive position map so that it contains the new secret permutation, π. The first time we perform a shuffle obliviously computing π is straightforward. Because the shuffle was performed with the composition of two Waksman networks as described previously, we can run the same network backwards using $(0,\ldots,n-1)$ as inputs to obtain π.

On subsequent shuffles, the process becomes complicated. The blocks in Shuffle are no longer in logical order because they have previously been shuffled and moved from Shuffle to Stash and back. Obtaining the permutation by the same method as above would require us to run both shuffles (four Waksman networks in total) in reverse, along with any other swaps that may have happened due to ordinary ORAM access. Each additional shuffle requires two more Waksman networks, and the number continues to increase without bound.

Instead, we augment each data block with a secret record of its logical index. When the blocks are shuffled, the logical indices are shuffled with them through the Waksman networks, and these indices comprise π^{-1}, the mapping from physical to logical index. To find π, the mapping from logical to physical, we simply invert π^{-1}.

To invert π^{-1} efficiently without allowing either party to learn anything about it, we adopt a technique from Damgård et al. [21]. The first party (Alice) locally samples a new random permutation π_a and computes the corresponding Waksman control bits. This is then used to jointly permute the elements of the secret

permutation π^{-1}, producing $\pi^{-1} \cdot \pi_a = \pi_b$. Next, π_b is revealed to the second party, Bob (but not to Alice). Bob does not learn anything about π^{-1} because it is masked by π_a. Bob now locally computes π_b^{-1}, and the two parties jointly execute another Waksman network to compute $\pi_a \cdot \pi_b^{-1} = \pi$.

4.5 Evaluation

To evaluate our design, we implemented our Square-Root ORAM design and Circuit ORAM, the best-performing previous ORAM design, using the same state-of-the-art MPC frameworks, and measured their performance on a set of microbenchmarks. We also wanted to understand the impact of different ORAM designs on application performance, and how close we are to enabling general-purpose MPC. To this end, we implemented several application benchmarks representing a wide range of memory behaviors and evaluated their performance with different ORAM designs.

4.5.1 Experimental Setup

We implemented and benchmarked RAM-SC protocols based on our ORAM as well as Circuit ORAM, using the Obliv-C [108] framework executing a Yao's garbled circuit protocol. Obliv-C provides a C-like language interface, and it incorporates many recent optimizations [10, 50, 109].

All code was compiled using gcc version 4.8.4, with the -O3 flag enabled. Unless otherwise specified, all reported times are wall-clock time for the entire protocol execution. Our benchmarks were performed with commercially available computing resources from Amazon Elastic Compute Cloud (EC2). We used compute-optimized instances of type C4.2xlarge running Amazon's distribution of Ubuntu 14.04 (64 bit). These notes provide four physical cores (capable of executing eight simultaneous threads in total), partitioned from an Intel Xeon E5-2666 v3, and 15 GiB of memory Our benchmarks are all single-threaded and cannot saturate the processing power available. We selected C4.2xlarge nodes on the basis of the greater bandwidth and memory they offer. Each benchmark was executed between two separate nodes within the same datacenter. We used iperf to measure the inter-node bandwidth, and found it to be about 1.03 Gbps.

In addition to square-root ORAM, we benchmarked a simple linear scan and an implementation of Circuit ORAM, the best previously reported ORAM construction for MPC. Our implementation of Circuit ORAM is much more efficient than the original implementation described in Wang et al. [101]. For example, while executing benchmarks on an Amazon C4.8xlarge EC2 instance for an ORAM of one million 32-bit blocks, they reported an access time of two seconds. On a less powerful, more bandwidth-constrained C4.2xlarge EC2 instance, our implementation requires only 0.16 seconds per access for an ORAM with the same parameters. This reduction by a factor of roughly twelve is mostly due to the efficiency advantages of the Obliv-C framework over the ObliVM [71] framework used by Wang et al.'s implementation. For all performance reported in the following, we let Circuit ORAM and square-root ORAM pack 8 entries in each recursive level. Circuit ORAM stops recursion when there are fewer than 2^8 entries.

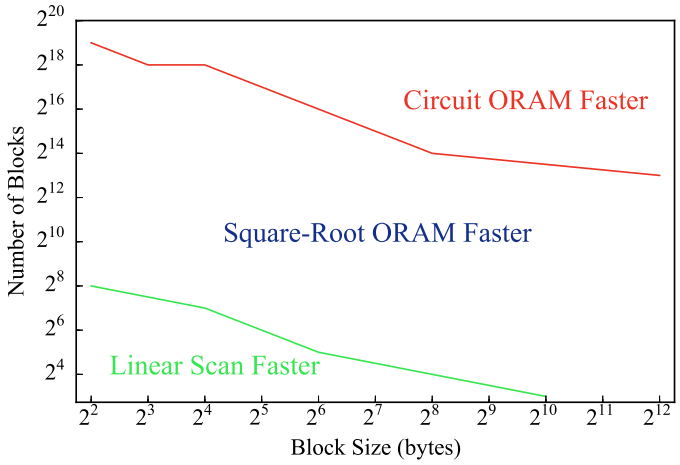


Figure 4.7: Per-access cost crossover points between ORAM schemes. Below the green line, linear scan is most efficient. Above the red line, Circuit ORAM is most efficient. Between the two, Square-Root ORAM is most efficient.

4.5.2 Microbenchmarks

We performed several microbenchmarks to assess the granular performance of different ORAM designs. We observed single-access execution time for block counts varying from 4 to 1024 and block sizes varying from 4 to 1024 bytes. This is the region of parameter space where the efficiencies of Square-Root ORAM and linear scan overlap. Figure 4.7 shows the efficiency crossover points derived from this data, ignoring initialization cost. Due to the nature of the Square-Root ORAM algorithm, each access is more expensive than the previous one, until a shuffle occurs and resets the cycle. To ensure our averages truly are representative, we collected a number of samples for each ORAM configuration equal to a multiple the shuffle period that is greater than thirty, except in the case of linear scan, for which exactly thirty samples were collected.

Breakeven points. Linear scan is preferred to Square-Root ORAM only for very small numbers of blocks. Circuit ORAM is orders of magnitude more expensive for similar parameters, due to its high fixed access cost. Our Square-Root ORAM implementation achieves a very low break-even point with linear scan. When using 4096 or fewer blocks, Circuit ORAM never wins over. And at a block size of 4 bytes, Circuit ORAM remains a suboptimal choice until we have more than 500,000 blocks. But that, in turn, increases initialization cost.

Comparison to Circuit ORAM. In comparing our Square-Root ORAM scheme to Circuit ORAM, we consider initialization and access costs separately since the number of accesses per initialization will vary across applications. Figure 4.8 shows the per-access wall-clock time for both designs, as well as for linear scan, ignoring initialization.

As expected, Circuit ORAM has the best asymptotic performance, but it also has a very high fixed cost per access, independent of the number of blocks. As a result, Square-Root ORAM performs better than Circuit

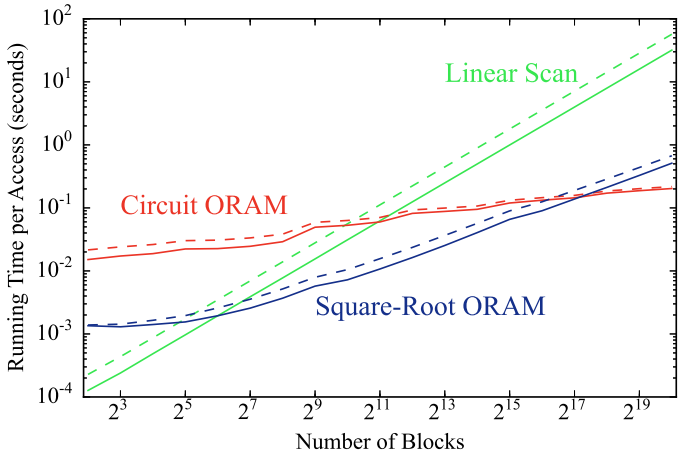


Figure 4.8: Cost per access omitting initialization. Solid lines are for block size of 16 bytes, dashed lines are for block size of 32 bytes. We collected a number of samples for each ORAM configuration equal to a multiple of the Square-Root ORAM shuffle period that is greater than thirty, except in the case of linear scan, for which exactly thirty samples were collected.

ORAM for all block counts up to 2^{16}, even ignoring initialization costs. In fact, for block counts less than $\tilde{}2^{11}$ linear scan also outperforms Circuit ORAM. These results are consistent with our analysis in Section 4.3.4 that Square-Root ORAM has worse asymptotic behavior, but smaller hidden constants.

For any application where the number of accesses is not significantly larger than the number of blocks in the ORAM, initialization cost must be considered. Figure 4.9 shows the initialization wall-clock times for Square-Root and Circuit ORAM, with parameters matching those in our access-time comparison. For this benchmark, we assume each ORAM must be populated using data already stored in an array of oblivious variables. In such a scenario, a linear scan ORAM requires only that the data be copied; the reported linear scan initialization speed is therefore equivalent to the time required to copy the data.

Initializing Square-Root ORAM is approximately 100 times faster than initializing Circuit ORAM, regardless of block count or block size. The standard way to populate Circuit ORAM is to insert each data element individually, using standard ORAM access operations; thus, the cost scales linearly with the number of blocks to be populated. We hypothesize that most of this speed improvement comes from having fewer network round trips in our initialization process. Circuit ORAM therefore requires $\Theta(N \log N)$ round trips for initialization, while our scheme requires only $\Theta(\log N)$.

4.5.3 Oblivious Binary Search

Unlike our other application benchmarks, binary search performs very few accesses relative to the ORAM size. An equivalent search can be performed using a single linear scan, and if only one search is to be performed, the linear scan is always more efficient. Consequently, we varied the number of searches performed for this benchmark, rather than the block size or block count. We benchmarked binary search using a block

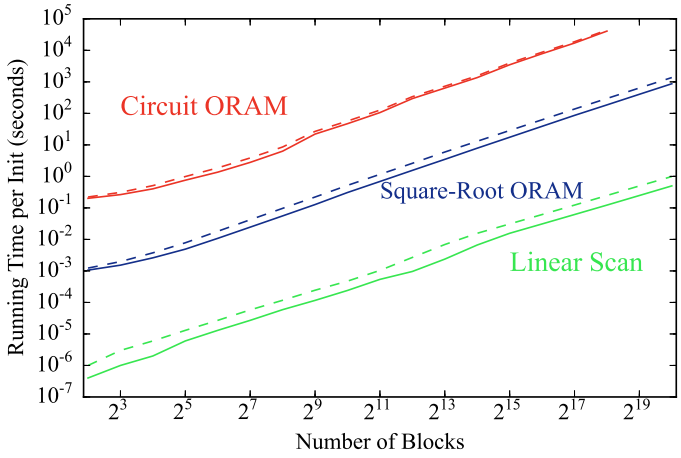


Figure 4.9: Cost of initialization. Solid lines are for block size of 16 bytes, dashed lines are for block size of 32 bytes.

size of 16 bytes and element counts of 2^{10} and 2^{15}. For arrays of 2^{10} elements, we averaged the running time over 30 samples, and for 2^{15} elements we use 3 samples. A few representative combinations for 2^{15} elements are reported in Table 4.1.

Initialization dominates execution time unless many searches are performed on the same data. As a result, Square-Root ORAM is more than two orders of magnitude better than Circuit ORAM when only one search is performed. For searches of 2^{10} elements, the linear scan method is more efficient than a binary search regardless of the ORAM type or the number of searches performed. Linear scan is initially faster for searches of 2^{15} elements as well, but Square-Root ORAM becomes more efficient than the linear scan method at 2^5 searches. Accesses to a Circuit ORAM of 2^{15} elements are more expensive than accesses to a Square-Root ORAM of the same size, so at this array size, Circuit ORAM will never be more efficient regardless of the number of searches performed.

4.5.4 Oblivious Breadth-First Search

Natively-oblivious formulations of Breadth-First Search (BFS) and other graph algorithms have been explored in the past [14]; however, we use a variant of the standard algorithm optimized for use in an oblivious context. It has complexity in $\Theta((V+E)C_{\mathsf{Access}})$, where C_{Access} is the complexity of accessing an element in the underlying ORAM construction. We allow our ORAM implementations to apply arbitrary functions to modify the blocks they access, as opposed to the simple read and write functions shown in Figure 4.2. This reduces the total number of ORAM accesses by, for example, permitting combined read and update operations. Rather than use an ORAM to house the queue, we use the oblivious queue data structure from Zahur and Evans [107].

We benchmarked our BFS implementation using linear scan, Circuit ORAM, and Square-Root ORAM.

Benchmark	Parameters	Linear Scan	Square-Root ORAM	Circuit ORAM
Binary Search	1 search	1.00	10.41	3228.69
	2^5 searches	31.87	26.25	3282.40
	2^{10} searches	1019.77	824.81	5040.82
Breadth-First Search	$n = 2^2$	0.09	0.34	4.28
	$n = 2^5$	4.77	4.08	42.66
	$n = 2^{10}$	4569.31	679.63	3750.57
Gale-Shapley	2^3 pairs	-	0.51	6.57
	2^6 pairs	-	145.13	1328.50
	2^9 pairs	-	119405.	188972.
Scrypt	$N = 2^5$	4.11	3.43	34.47
	$N = 2^{10}$	1678.16	293.79	1453.85
	$N = 2^{14}$	*about 7 days*	1919.92	2846.51
	Litecoin	210.92	40.29	247.29

Table 4.1: Summary of benchmark results. All benchmark results are average measured wall-clock time in seconds for full protocol execution (see individual benchmark sections for details).

We took 30 samples for experiments of n vertices and $\gamma \times n$ edges, with n ranging from 4 to 1024 and γ as 8. For each sample, a fresh set of edges were generated randomly among the chosen number of vertices. A few representative combinations are shown in Table 4.1.

The results of the BFS benchmark roughly follow the pattern established by the microbenchmarks in Section 4.5.2. Small numbers of vertices and edges yield small ORAMs, and linear scan proves to be best in these cases. As the number of vertices or edges begins to rise, Square-Root ORAM quickly becomes more efficient than linear scan. Our BFS implementation uses blocks of only a few bytes each; as a result, Circuit ORAM eventually becomes more efficient than linear scan, but it does not approach the efficiency of Square-root ORAM before the upper bound of our testing range is reached at $n = 2^{10}$. Beyond that point the benchmarks would have required several hours to complete.

4.5.5 Oblivious Stable Matching

To explore a benchmark representative of a complex algorithm, we implemented an oblivious version of the Gale-Shapley stable matching algorithm [31]. We followed the textbook algorithm closely. Although we believe there are significant optimizations available in adapting the algorithm for use in MPC, they are beyond the scope of this work.

As a result, our implementation requires $\Theta(n^2)$ accesses of an ORAM with n^2 elements. It also uses of several ORAMs of length n. The most efficient arrangement may be to mix ORAM schemes, but we have not done this. As in our BFS implementation, we used function application to reduce the number of ORAM accesses.

We benchmarked our implementation of Gale-Shapley with both Circuit and Square-Root ORAMs as the

underlying structure, but not linear scan since it is clear linear scan cannot be competitive for this benchmark and the expense of executing it on non-trivial sizes would be considerable. The number of pairs to be matched ranged from 4 to 512. When the pair count was less than 128, we collected 30 samples; for pair counts of 128 and 256 we collected 3 samples; for 512 pairs, we collected one sample. Results for few representative configurations are included in Table 4.1.

Square-root ORAM proved more efficient over the entire range we benchmarked, although for sufficiently large sizes Circuit ORAM will eventually do better. For 64 pairs, Square Root ORAM is over 9 times faster (finishing in 145 seconds); for 512 pairs, stable matching requires just over 33 hours using Square-Root ORAM and 52.5 hours with Circuit ORAM.

4.5.6 Oblivious Scrypt

To explore the possibility of using ORAMs in a challenging cryptographic application, we implemented the key derivation function *scrypt* [82]. Scrypt was originally intended to be difficult to parallelize, and therefore difficult to break by brute force, even with custom high performance hardware. It achieves this goal by repeatedly enciphering a single block of data, retaining each intermediate result in memory. It then performs a second round of encipherment, mixing the block with an intermediate result from the first round selected according to the current value. In an oblivious context, scrypt *requires* the use of an ORAM of some sort, as the indices of the memory accesses in the second phase depend upon oblivious data generated in the first phase. Due to its unpredictable memory access pattern, the scrypt algorithm is designed to require sequential execution with no significant shortcuts.

With typical parameters, scrypt requires a relatively small ORAM element count. For instance, Litecoin, which uses scrypt as a cryptocurrency proof-of-work, specifies $N = 2^{10}$ elements [69], and Colin Percival, the designer of scrypt, recommends a minimum of $N = 2^{14}$ elements for normal use [82]. On the other hand, Percival recommends that each element be 1KB in size — much larger than required by any of our other application benchmarks. In the course of execution, scrypt performs exactly one access per element.

We tested scrypt using the recommended parameters and test vectors from the scrypt specification [82], $r = 8$ and $p = 1$, and we varied N from 4 to 2^{14}. In addition, we benchmarked the parameters used by Litecoin, ($r = 1$, $p = 1$, $N = 2^{10}$). A few representative combinations are presented in Table 4.1. As in the other benchmarks, linear scan is marginally more efficient when the number of blocks (N) is small. Otherwise, Square-Root ORAM is preferred; it exceeds the performance of linear scan by approximately one order of magnitude when $N = 2^{10}$, and this ratio improves as N increases.

The largest parameters we benchmarked are Percival's recommended minimum parameters ($r = 8$, $p = 1$, $N = 2^{14}$), which he originally chose on the basis that they required less than 100ms to execute on contemporary hardware, this being what he considered a reasonable threshold for interactive use [82]. On our EC2 test node, the reference (non-oblivious) scrypt implementation requires 35ms with the same parameters. With Square-Root ORAM as the underlying primitive, execution took 32 minutes, compared with 47 minutes for Circuit ORAM. The large block size required by scrypt causes block access time to form a greater portion

of the total cost than in our other application benchmarks. As a result, Circuit ORAM becomes competitive earlier than in the other cases. We did not benchmark linear scan for the recommended parameters; we estimated that it would require roughly 7 days to complete, well beyond what could reasonably be considered useful in practice.

Even with Square-root ORAM, scrypt requires 55,000 times longer to execute with real-world parameters as an MPC protocol than it does to execute conventionally. This is almost certainly too expensive to be practical for any interactive application today, but shows that even complex algorithms designed intentionally to be expensive to execute are not beyond the capabilities of general-purpose MPC today.

4.6 Conclusion

The success of MPC depends upon enabling developers to create efficient privacy-preserving applications, without requiring excessive effort, expertise, or resources. It is important that MPC protocols be compatible with conventional programming techniques and data structures with depend on random access memory. Our Square-Root ORAM design provides a general-purpose oblivious memory that can be used anywhere a programmer would normally use an array. We have presented a new approach for designing ORAMs for MPC, which demonstrates how hierarchical ORAM designs can be implemented efficiently, and how they can overcome the high initialization costs and parameter restrictions of previous ORAM designs. This represents a step towards a programing model for MPC in which standard algorithms can be efficiently implemented as MPCs, using oblivious memory just like conventional memory is used today.

Chapter 5

Conclusion

The purpose of this work is to present new constructions that allow MPC programs to efficiently perform data-dependent memory accesses, allowing efficient implementations of various interesting applications in MPC. Part of this was also facilitated by a new programming language to support the development of efficient protocols.

5.1 Summary

We developed a new programming language, Obliv-C, that allows programmers to easily write MPC programs without any cryptographic expertise. In particular, the design takes special care to integrate with existing ecosystem of C libraries, profiling and debugging tools. We have further demonstrated the versatility of our approach by implementing ORAM, some of the circuit structures, and several utility libraries and benchmarks. This approach also allows researchers to easily try new protocols and techniques across many benchmark applications without requiring them to modify the compiler.

We note that in the past, most demonstrations of MPC had to be limited to simple applications like matrix multiplication or AES encryption, where memory access pattern never depended on sensitive inputs. We resolve this problem by designing an ORAM scheme that provides concrete efficiency at reasonable sizes. We also provide efficient constructions for primitive data structures that can be easily used by application developers without being experts in circuit design.

5.2 Conclusion

Let's consider once again the example that we started with: two hospitals wishing to jointly analyze their patient records. Today, we are at a point where such programs can be written by developers without any cryptographic expertise. MPC execution speed has improved to the stage where we can find intersection

between sets of a million records in a matter of minutes. In the past, there was no way to perform such computation in a privacy-preserving manner. If a developer is looking for ways to incorporate MPC into their applications, they now have a body of work that they can use.

We still have many challenges remaining in this area. There are important questions still open about how MPC will be used in practice. For example, given a function being computed securely, there is no easy way to gauge if the output by itself is revealing "too much" about private inputs. In other words, in general it is hard to know if an adversary can use the output to learn sensitive information. Additionally, most of our experiments are still in the honest-but-curious model, where we assume that the other parties faithfully follow the protocol. Fully malicious protocols are still about an order of magnitude slower. There is also the big question of whether the rapid performance improvements we have witnessed over the last decade will continue. It is possible that we are reaching performance limits and it will become increasingly difficult to make orders of magnitude improvements in future.

Finally, there is a very serious question of how much we trust developers of MPC applications. The entire premise of MPC is that, for some applications, it is too difficult to find a third party trusted by all entities interested in the computation. Currently, however, we are implicitly placing our trust in a single cohort of compiler writers. In the future, we need to find ways to have frameworks independently verified, or at even have independently developed MPC frameworks interoperate, so that no single party is trusted by everyone.

In spite of these challenges, we are already starting to see the use of MPC in practice. We have already seen its use in auctions and statistical applications — and it's only been six or so years that large MPC applications have become feasible. So we are confident that the trend of widespread MPC will continue, and that we will see the development of new kinds of applications that were not previously considered. At the same time, we expect to see a corresponding interest from the research community in reducing MPC performance overheads. Hopefully this symbiotic relationship will accelerate the widespread adoption of MPC as a standard tool in the development of everyday applications that work on private data.

Bibliography

[1] The zettabyte era: Trends and analysis. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.pdf. Accessed: 21st March, 2016.

[2] Advanced encryption standard (aes) instructions set. http://www.ferretronix.com/unigroup/intel_aes_ni/aes-instructions-set_wp.pdf, 2008. Accessed: 21st March, 2016.

[3] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to Efficiently Evaluate RAM Programs with Malicious Security. In *EUROCRYPT*, 2015.

[4] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[5] A.E. Andreev. On a method for obtaining lower bounds for the complexity of individual monotone functions. In *Soviet Math. Dokl*, volume 31, 1985.

[6] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly-secure multiparty computation. Cryptology ePrint Archive, Report 2011/136, 2011. http://eprint.iacr.org/2011/136.

[7] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *ACM Conference on Computer and Communications Security*, 2013.

[8] Domagoj Babic and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008.

[9] Ken E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968.

[10] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE

Computer Society Press, May 2013.

[11] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[12] Armin Biere. Lingeling and friends at the sat competition 2011. *FMV Report Series Technical Report*, 11(1), 2011.

[13] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward. In *ACM Conference on Computer and Communications Security*, 2015.

[14] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ACM Symposium on Information, Computer and Communications Security*, 2013.

[15] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis. *Financial Cryptography and Data Security*, 2012.

[16] Peter Bogetoft, Dan Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Nielsen, Jesper Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. *Financial Cryptography and Data Security*, 2009.

[17] Ravi B. Boppana. Amplification of probabilistic boolean formulas. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*. IEEE, 1985.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[19] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2), 2008.

[20] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. *ACM SIGSOFT Software Engineering Notes*, 26(5), 2001.

[21] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, March 2006.

[22] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology—CRYPTO*. 2012.

[23] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[24] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: The Secure Computation Application Programming Interface. *IACR Cryptology ePrint Archive*, 2012.

[25] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data mining*, volume 1996. AAAI Press, 1996.

[26] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-Party ORAM for Secure Computation. In *ASIACRYPT*, 2015.

[27] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *ACM Sigplan Notices*, volume 37. ACM, 2002.

[28] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient Secure Two-Party Computation from General Assumptions. In *Advances in Cryptology—EUROCRYPT*. 2013.

[29] Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and Maliciously Secure Two-party Computation using the GPU. In *Applied Cryptography and Network Security*, 2013.

[30] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology — EUROCRYPT 2004*, 2004.

[31] David Gale and Lloyd S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[32] Vijay Ganesh and David Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*. Springer, 2007.

[33] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[34] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using it Efficiently for Secure Computation. In *Privacy Enhancing Technologies*, 2013.

[35] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private Database Access with HE-over-ORAM Architecture. In *Applied Cryptography and Network Security*, 2015.

[36] Patrice Godefroid. Compositional dynamic test generation. In *ACM SIGPLAN Notices*, volume 42. ACM, 2007.

[37] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[38] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *ACM Sigplan Notices*, volume 45. ACM, 2010.

[39] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[40] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3), 1996.

[41] Shafi Goldwasser, Silvio M. Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with an honest majority. In *Proc. of the Nienteenth Annual ACM STOC*, volume 87, 1987.

[42] Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010.

[43] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part II*, volume 6756 of *LNCS*, pages 576–587. Springer, July 2011.

[44] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In Yuval Rabani, editor, *23rd SODA*, pages 157–167. ACM-SIAM, January 2012.

[45] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 513–524. ACM Press, October 2012.

[46] Trevor Hansen, Peter Schachte, and Harald Søndergaard. State joining and splitting for the symbolic execution of binaries. In *Runtime Verification*. Springer, 2009.

[47] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis,

and Vitaly Shmatikov, editors, *ACM CCS 10*, pages 451–462. ACM Press, October 2010.

[48] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 772–783. ACM Press, October 2012.

[49] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[50] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *20th USENIX Security Symposium*, 2011.

[51] Yan Huang, Jonathan Katz, and David Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012.

[52] Yan Huang, Lior Malka, David Evans, and Jonathan Katz. Efficient privacy-preserving biometric identification. In *Network and Distributed System Security Symposium*, 2011.

[53] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. GPU and CPU Parallelization of Honest-but-Curious Secure Two-Party Computation. In *ACM Annual Computer Security Applications Conference*, 2013.

[54] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending Oblivious Transfers Efficiently. In *Advances in Cryptology—CRYPTO*, 2003.

[55] Stanislaw Jarecki and Vitaly Shmatikov. Efficient Two-Party Secure Computation on Committed Inputs. In *Advances in Cryptology—EUROCRYPT*, 2007.

[56] Marcel Keller and Peter Scholl. Efficient, Oblivious Data Structures for MPC. In *ASIACRYPT*, 2014.

[57] V.M. Khrapchenko. The complexity of the realization of symmetrical functions by formulae. *Mathematical Notes*, 11(1), 1972.

[58] Sarfraz Khurshid, Corina PĂsĂreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[59] Peter Kogge and John Shalf. Exascale computing trends: Adjusting to the. *Computing in Science & Engineering*, 2013.

[60] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. *Cryptology and Network Security*, 2009.

[61] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. *Automata, Languages and Programming*, 2008.

[62] Ben Kreuter, Benjamin Mood, abhi shelat, and Kevin Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *22nd USENIX Security Symposium*, August 2013.

[63] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 285–300. USENIX Association, 2012.

[64] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *23rd SODA*, pages 143–156. ACM-SIAM, January 2012.

[65] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2012.

[66] Yehuda Lindell and Benny Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Advances in Cryptology—EUROCRYPT*. 2007.

[67] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, 22(2), 2009.

[68] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 329–346. Springer, March 2011.

[69] Litecoin Project. scrypt.cpp. `https://github.com/litecoin-project/litecoin/blob/master-0.10/src/crypto/scrypt.cpp`, 2015.

[70] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating Efficient RAM-Model Secure Computation. In *IEEE Symposium on Security and Privacy*, 2014.

[71] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A Programming Framework for Secure Computation. In *IEEE Symposium on Security and Privacy*, 2015.

[72] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 377–396. Springer, March 2013.

[73] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In *USENIX Security Symposium*, 2004.

[74] William Melicher, Samee Zahur, and David Evans. An intermediate language for garbled circuits (poster abstract). IEEE Symposium on Security and Privacy, 2012, May 2012.

[75] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation.

Public Key Cryptography-PKC 2006, 2006.

[76] Benjamin Mood, Lara Letaw, and Kevin Butler. Memory-efficient Garbled Circuit Generation for Mobile Devices. In *Financial Cryptography and Data Security*. 2012.

[77] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and kdcs. In *Advances in CryptologyEUROCRYPT99*. Springer, 1999.

[78] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Conference on Compiler Construction*, 2002.

[79] Jesper B. Nielsen, Peter S. Nordholt, Claudio Orlandi, and Sai S. Burra. A new approach to practical active-secure two-party computation. *Advances in Cryptology–CRYPTO 2012*, 2012.

[80] Jesper Buus Nielsen and Claudio Orlandi. LEGO for Two-Party Secure Computation. In *Theory of Cryptography Conference*, 2009.

[81] Jan Obdržálek and Marek Trtík. Efficient loop navigation for symbolic execution. *Automated Technology for Verification and Analysis*, 2011.

[82] Colin Percival. Stronger key derivation via sequential memory-hard functions. `http://www.tarsnap.com/scrypt/scrypt.pdf`, 2009.

[83] B. Pinkas, T. Schneider, N.P. Smart, and S. Williams. Secure two-party computation is practical. Cryptology ePrint Archive, Report 2009/314, 2009. `http://eprint.iacr.org/2009/314`.

[84] Benny Pinkas. Cryptographic techniques for privacy-preserving data mining. *ACM SIGKDD Explorations Newsletter*, 4(2):12–19, 2002.

[85] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 502–519. Springer, August 2010.

[86] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. In *23rd USENIX Security Symposium*, 2014.

[87] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *Journal of the ACM (JACM)*, 26(2), 1979.

[88] Corina S. Psreanu, Peter C. Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008.

[89] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. 2014.

[90] Phillip Rogaway and John P. Steinberger. Constructing cryptographic hash functions from fixed-key blockciphers. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 433–450. Springer, August 2008.

[91] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009.

[92] Axel Schropfer, Florian Kerschbaum, and Gunter Muller. L1-an intermediate language for mixed-protocol secure computation. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*. IEEE, 2011.

[93] Claude Shannon. A symbolic analysis of relay and switching circuits. MIT Master's Thesis, 1937.

[94] abhi shelat and Chih-hao Shen. Two-Output Secure Computation with Malicious Adversaries. In *Advances in Cryptology—EUROCRYPT*, 2011.

[95] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, December 2011.

[96] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 299–310. ACM Press, November 2013.

[97] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic, Part II*, 1968.

[98] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. *Advances in Cryptology–EUROCRYPT 2010*, 2010.

[99] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2), 1996.

[100] Abraham Waksman. A Permutation Network. *Journal of the ACM*, 15(1), January 1968.

[101] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM Conference on Computer and Communications Security*, 2015.

[102] Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure Computation of MIPS Machine Code. Cryptology ePrint Archive, Report 2015/547, 2015. `http://eprint.iacr.org/2015/547`.

[103] Xiao Shaun Wang, Yan Huang, TH Chan, Abhi Shelat, and Elaine Shi. Scoram: Oblivious ram for secure computation. In *ACM Conference on Computer and Communications Security*. ACM.

[104] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 293–304. ACM Press, October 2012.

[105] Yan Huang and Jonathan Katz and David Evans. Quid Pro Quo-tocols: Strengthening Semi-Honest Protocols with Dual Execution. In *33rd IEEE Symposium on Security and Privacy*, 2012.

[106] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, 1982.

[107] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *34th IEEE Symposium on Security and Privacy*, 2013.

[108] Samee Zahur and David Evans. Obliv-C: A Lightweight Compiler for Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153, 2015. `http://oblivc.org`.

[109] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *EUROCRYPT*. 2015.

[110] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A General-Purpose Compiler for Private Distributed Computation. In *ACM Conference on Computer and Communications Security*, 2013.