The Oblivious Machine
or: How to Put the C into MPC

Marcel Keller

Department of Computer Science, University of Bristol
m.keller@bristol.ac.uk

Abstract. We present an oblivious machine, a concrete notion for a multiparty random access machine (RAM) computation and a toolchain to allow the efficient execution of general programs written in a subset of C that allows RAM-model computation over the integers. The machine only leaks the list of possible instructions and the running time. Our work is based on the oblivious array for secret-sharing-based multiparty computation by Keller and Scholl (Asiacrypt '14). This means that we only incur a polylogarithmic overhead over the execution on a normal CPU.
We describe an implementation of our construction using the Clang compiler from the LLVM project and the SPDZ protocol by Damgård et al. (Crypto '12). The latter provides active security against a dishonest majority and works in the preprocessing model. The online phase clock rate of the resulting machine is 41 Hz for a memory size of 1024 64-bit integers and 2.2 Hz for a memory of 2^{20} integers. Both timings have been taken for two parties in a local network. Similar work by other authors has only been the semi-honest setting.
To further showcase our toolchain, we implemented and benchmarked private regular expression matching. Matching a string of length 1024 against a regular expression with 69270 transitions as a finite state machine takes seven hours online time, of which more than six hours are devoted to loading the reusable program.

Keywords: Multiparty computation, random-access machine, oblivious RAM, compilers, regular expression matching

1 Introduction

Multiparty computation (MPC) refers to a technique that allows a set of parties to compute on data held by them privately without revealing anything to each other, bar the desired result. The feasibility has been established for some time in two lines of work, Yao's garbled circuits [16] and secret-sharing-based multiparty computation [1, 4, 11]. The former allow two parties to compute binary circuits, and the latter enables any number of parties to compute arithmetic circuits over finite fields or rings. In this paper, we focus on secret-sharing-based MPC. There are various schemes differing in the degree of adversarial power such as the number of corrupted parties or the kind of corruption. However, all of them implement the so-called arithmetic black box presented in Section 2.

While circuits are complete in the sense that they allow any computation, they generally incur an overhead over random access machine (RAM) programs. This overhead is related to the fact that, to access an array by a data-dependent index, a circuit needs to access the whole array. In addition, accessing only parts of such an array would reveal possibly sensitive data. A first step to remedy this was taken by Ostrovsky and Shoup [21], who proposed the oblivious random access machine (ORAM) as a mean to hide the access pattern of a memory-restricted client on a server with larger memory. They briefly mention the possibility of using their scheme in the context of secure two-party computation with one party holding the encrypted server memory. Damgård et al. [7] on the other hand suggested to secret share the server memory. However, due to the lack of efficient ORAM schemes, no concrete schemes or implementations emerged.

Only following the proposal of tree-based ORAM by Shi et al. [24], practical instantiations of oblivious data structures for multiparty computation have been proposed, both for Yao's garbled circuits [12, 17, 29] and secret-sharing based MPC [13]. The former only provide security against a semi-honest adversary, while

the latter does so against a malicious adversary. These works essentially provide an implementation of an oblivious array with efficient access, that is, one access to a secret index only incurs polylogarithmic cost (in the size of the array). Based on the oblivious array, the latter work goes on to implement an oblivious priority queue, which is then used for Dijkstra's algorithm, as well as the Gale-Shapley algorithm for stable matching. In the case of Dijkstra's algorithm, it turns out that the algorithm has to be reformulated to be implemented as a circuit with access to oblivious data structures.

On the theoretical side, Gentry et al. [10] proposed garbled RAM, which combines Yao's garbled circuit and ORAM. They present two solutions, one based on identity-based encryption, and the other based on revocable pseudorandom functions. Both approaches do not seem to be as practical because they involve the mentioned cryptographic operations being executed in a garbled circuit. In comparison, the works presented in the previous paragraph use relatively lightweight operations.

1.1 Our Contribution

In this work, we present a practical instantiation of an oblivious machine in the arithmetic black-box model, that is, an actively secure MPC protocol that allows efficient, oblivious computation in the RAM model. By oblivious computation we mean that the sequence of instructions executed is not revealed to the adversary, only the running time is. This enables the compilation of a subset of ANSI C (including conditional expressions, loops, arrays, and structs) and thus the execution of many algorithms in C with only polylogarithmic overhead. To the best of our knowledge, we are the first to implement this. We also present a theoretical model of an oblivious machine.

While previous works [12, 17] have introduced the concept of secure RAM-model computation, their notions remain rather abstract. Furthermore, Liu et al. [17] call general secure RAM-model computation "relatively inefficient" because one has to execute the universal next-instruction circuit, which must interpret every possible instruction. By contrast, the motivation of this work is to put a price tag on such general secure RAM-model computation.

Our construction essentially uses the oblivious array by Keller and Scholl [13] for storing data and code, and for every step, it executes all possible instructions in a way that minimizes data accesses. While this incurs some overhead, we believe that it is more efficient than using a one instruction set machine because such a machine will inevitably increase the length of programs and thus the length of memory accesses, which we have found to be the most expensive part in our implementation.

As an application of our concept, we highlight the case of regular expression matching. Regular expressions can be implemented as finite state machines and thus in the RAM model. Kerschbaum [14] presented two MPC protocols for regular expression matching, a secure one and one with leakage. They have complexity in $O(nml)$ and $O(knm^2 + ln)$ for m states, n symbols, string length l and some security parameter k. The security of our solution lies in between because it only leaks the total running time, which is less than every repetition of a previous state being leaked by Kerschbaum's algorithm. With a complexity in $O(nm \log^3 nm + ln(\log^3 nm + \log^3 l))$, our approach beats the previous work on regular expressions that are complex enough, that is, if $km \gg \log^3 nm$ and $km^2 \gg l(\log^3 nm + \log^3 l)$. Using the same approach using a regular CPU has complexity in $O(nm + ln)$ for loading and executing the program. Launchbury et al. [15] also mention an implementation of regular expression matching with multiparty computation. From their description, we estimate that their protocol is similar to the secure one by Kerschbaum.

With MPC, potentially corrupted parties are involved in any computation. While the oblivious machine obscures the instruction currently executed as well as the data accessed, it is inherent that the adversary learns the amount of computation. In our case, this is the number of instructions being executed. A straightforward way to obscure this to some extent is to define a maximal number of instructions and then execute exactly this many steps. However, this can only increase the computation time. We believe that there still is a use case for oblivious computation leaking the total time, for example, if the same program is not executed enough times to mount a timing attack.

Since our oblivious machine does not reveal information about the code other than the set of possible instructions, and code can also be input in secret, it also suits private function evaluation (PFE). Informally, private function evaluation allows two parties to compute a function known to one party on data known by

another party without revealing either input to the other party. Previous work on PFE focuses on circuits, such as the solution by Mohassel et al. [20]. Their solution only incurs a constant overhead for circuits. While our solution for RAM-model computation comes with polylogarithmic overhead, it is the first such proposal to the best of our knowledge.

1.2 Related Work

Keller and Scholl [13], while providing much of the foundation of our work, do not consider general oblivious computation but stick to oblivious data structures and specific applications thereof.

Similarly, SCVM [18] and ObliVM [19] are two-party computation implementations that use ORAM for oblivious arrays, but they do not fully support the RAM model. For example, when branching on secret variables, both branches are executed. This makes these approaches infeasible to use with programs using GOTO statements such as the ones output by the regular expression compiler descriped in Section 5. Furthermore, both SCM and ObliVM do not hide the program being executed.

Wang et al. [29] briefly mention the idea of implementing a universal RAM instruction as a circuit. However, they do not present a more detailed account or experimental figures. In a recent preprint [28], Wang et al. propose a compiler of bytecode for a particular processor (MIPS) to garbled circuits. Their system falls short in comparison to ours in two aspects. First, they analyze a program to find out at which time in the execution a memory access might be necessary. While this reduces the number of expensive ORAM computation, this is limited to relatively small programs because it requires the computation of every possible execution path of the program. In comparison, our approach works for every program because it allows memory accesses in every step. Secondly, their garbled circuit implementation only provides semi-honest security compared to malicious security in our case. The latter also explains the offline phase that is about 1000 times more expensive than the online phase. This is inherent to the SPDZ protocol.

Our oblivious machine is related to the concept of an oblivious Turing machine, which is a Turing machine where the movement of the head only depends on the time. Pippenger and Fischer [22] showed that any Turing machine can be converted into an oblivious one incurring only logarithmic overhead. However, the best known result for converting a RAM program with running time T to a Turing machine results in a running time in $O(T^2)$ [5]. Hence, this transformation is currently not suitable to achieve polylogarithmic overhead for RAM programs.

In the area of homomorphic encryption, Gentry et al. [9] have proposed to use ORAM to enable private queries to an encrypted database. They do not target general computation however.

Ben-Sasson et al. [2] proposed TinyRAM, a system for succinct verifiable non-interactive arguments to prove the correct execution of C programs. The setting of verifiable computation differs from ours in that the control flow does not need to be hidden. This is what mandates the use of oblivious RAM in our construction.

1.3 Paper Organization

After introducing the concept of MPC, oblivious arrays in MPC, and the RAM model in Section 2, we present a model of our oblivious machine in Section 3. We also prove that this model implements RAM computation, and that it can be implemented using MPC with oblivious arrays. In Section 4, we the present our actual implementation and toolchain with examples in Section 5 on the application to regular expression matching. Finally, we show experimental results in Section 6, and we conclude in Section 7.

2 Preliminaries

In this section, we will present previous results that are related to our work. We start with multiparty computation. MPC schemes allow a set of parties to compute public circuits on private data. All secret-sharing-based MPC protocols provide the arithmetic black box shown in Figure 1. Since most schemes use linear secret sharing where the sharing commutes with linear operations, the addition operation can

Initialize: On input $(\mathsf{Init}, \mathbb{F})$ from all parties, store $\mathbb{F}$.
Public Input: On input $(\mathsf{PublicInput}, a)$ from all parties, store a and return a handle $[a]$ to all parties.
Private Input: On input $(\mathsf{PrivateInput}, i, a)$ from Party P_i and $(\mathsf{PrivateInput}, i)$ from all other parties, store a and return a handle $[a]$ to all parties.
Addition: On input $(\mathsf{Add}, [a], [b])$ from all parties, compute $c = a + b$ in $\mathbb{F}$, store it, and return a handle $[c]$ to the parties.
Multiplication: On input $(\mathsf{Multiply}, [a], [b])$ from all parties, compute $c = a \cdot b$ in $\mathbb{F}$, store it, and return a handle $[c]$ to the parties.
Public Output: On input $(\mathsf{PublicOutput}, [a])$ from all parties, reveal a to all parties.
Private Output: On input $(\mathsf{PrivateOutput}, i, [a])$ from all parties, reveal a to Party P_i.
Abort: On input Abort from the adversary, abort.

Fig. 1. $\mathcal{F}_{ABB}$

Initialize Array: On input $(\mathsf{initArray}, n)$ from all parties, allocate an array of size n, initialize its entries to zero, and return a handle $[x]$ to all parties.
Read Array: On input $(\mathsf{readArray}, [x], [a])$ from all parties, read the a-th entry of x, store it, and return a handle $[b]$ to it.
Write Array: On input $(\mathsf{writeArray}, [x], [a], [v], [f])$ from all parties, write v to the a-th entry of x if $f = 1$.

Fig. 2. $\mathcal{F}_{ABBOA}$ extension

often be done without communication. On the other hand, multiplication cannot be computed without communication.

Note that the arithmetic black box does not specify the security properties achieved by a particular protocol, for example, whether it allows active or passive corruption, how many parties can be corrupted etc. This goes beyond the scope of the theoretical part of this paper because the security depends on the protocol. The only property specified by $\mathcal{F}_{ABB}$ is the possibility for the adversary to abort the protocol. This is required for protocols that allow a malicious, dishonest majority such as as the SPDZ protocol [8]. However, our protocols can be instantiated with any MPC scheme implementing $\mathcal{F}_{ABB}$, and the resulting protocol will inherit the security properties of the underlying scheme. Furthermore, because $\mathcal{F}_{ABB}$ only outputs handles like $[a]$ to intermediate information, it is often straight-forward to prove the security of protocols using it.

Keller and Scholl [13] proposed to use tree-based ORAM in the context of $\mathcal{F}_{ABB}$ to get oblivious arrays with polylogarithmic overhead. Their result can be used to extend $\mathcal{F}_{ABB}$ to $\mathcal{F}_{ABBOA}$ as detailed in Figure 2. The proposed construction keeps both the client and the server memory of an ORAM scheme in the arithmetic black box, executes client computations using $\mathcal{F}_{ABB}$ addition and multiplication, and uses the public output of $\mathcal{F}_{ABB}$ to reveal the address for server memory accesses (and hence the $\mathcal{F}_{ABB}$ handle). Their result can be stated as $\mathcal{P}_{ABBOA}$ in Figure 3.

Theorem 1. *$\mathcal{P}_{ABBOA}$ realizes $\mathcal{F}_{ABBOA}$ in the $\mathcal{F}_{ABB}$-hybrid model.*

Proof (Sketch). We simulate the ORAM operations using an emulation of $\mathcal{F}_{ABB}$ similarly to the protocol as follows: We use the ORAM simulator to generate the server memory addresses being revealed and abort $\mathcal{F}_{ABB}$ in case of deviation by corrupted parties. The ORAM simulator guarantees the sequence of addresses accessed in the server memory are independent of the access pattern. Furthermore, the probability of the ORAM delivering incorrect data is negligible. Hence, the simulation is indistinguishable from the protocol.

2.1 Random Access Machines

We will use the RAM model by Cook and Reckhow [5]. They define a random access machine as a machine with a memory X of integer registers that can execute a finite program consisting of the instructions listed in Table reftab:ram. The first instruction allows to load a constant to a fixed memory address, the second and

$\mathcal{F}_{ABB}$ **instructions:** Use $\mathcal{F}_{ABB}$ as instructed.
Initialize Array: Allocate sufficient storage according the ORAM scheme and run the ORAM initialization using $\mathcal{F}_{ABB}$.
Read Array: Read the address stored in x at a using $\mathcal{F}_{ABB}$ as outlined above and return the resulting handle as $[b]$.
Write Array: Read the address stored in x at a as above, and use the ORAM protocol to store $\mathsf{IfElse}(f, v, b) = f \cdot (v - b) + b$ at address a in x.

Fig. 3. $\mathcal{P}_{ABBOA}$

third implement addition and subtraction with fixed input and output addresses. The next two instructions enable indirect addressing for loading and storing, which is required for array operations, for example. The "TRA" instruction (for transfer) allows to jump conditionally in the program code, which is needed by all control flow operations such as "if" statements and loops. Finally, "READ" and "WRITE" cover input and output operations.

$X_i \leftarrow C$	C any integer
$X_i \leftarrow X_j + X_k$	
$X_i \leftarrow X_j - X_k$	
$X_i \leftarrow X_{X_j}$	
$X_{X_i} \leftarrow X_j$	
TRA m if $X_j > 0$	Jump to address m in the program code
READ X_i	Read from input
PRINT X_i	Print to output

Table 1. Instructions of random access machines

3 The Oblivious Machine

In this section, we will present our theoretical oblivious machine, prove that it implements RAM-model computation, and prove that it can implemented using the arithmetic black box with oblivious arrays. Figure 4 shows the desired functionality of the oblivious machine.

For simplicity, we only allow either public code or private code. One could think of mixing those, but this would require some intricate linking. Similarly, we decided to separate instruction and data storage to simplify the description. This results in a Harvard-like architecture, where the memory for instructions is separated from the data storage. It is straight-forward to use a Von Neumann architecture, where everything is stored in the same memory. However, due to the polylogarithmic access complexity of the underlying ORAM, two smaller oblivious arrays are slightly faster than one combined. Furthermore, code and data have different formats in our implementation, which is easier to accommodate for in two different oblivious arrays. Finally, we do not see a use case for code that can be altered by the machine.

We now present an abstraction that allows to minimize the number of memory accesses per execution step. Essentially, we require all possible instructions to have the same pattern with respect to memory accesses. This allows to execute all instructions in parallel, as required by design, while accessing the memory independently of the number of possible instructions. We formalize the access pattern as an instruction pattern, which also includes the size of the state that is carried across a memory access. An instruction is with a certain instruction pattern is defined by the circuits that are computed before, inbetween, and after the memory accesses. The first circuit uses the runtime arguments param as input state, while the last circuit outputs the program counter of the next instruction to be executed. The circuits also are given the value

Initialize: On input $(\mathsf{Init}, \mathbb{F}, n_d)$ from all parties, store the parameters and allocate the data array of size n_d.

Instructions: On input $(\mathsf{Instructions}, Q, I_0, \ldots, I_{n_I})$ from all parties, store the list of instructions and the instruction pattern Q.

Public Code: On input $(\mathsf{PublicCode}, (c_0, \mathsf{param}_0), \ldots, (c_{n_c-1}, \mathsf{param}_{n_c-1}))$ from all parties, store the code array.

Private Code: On input $(\mathsf{PrivateCode}, i, (c_0, \mathsf{param}_0), \ldots, (c_{n_c-1}, \mathsf{param}_{n_c-1}))$ from Party P_i and $(\mathsf{privateCode}, i, n_c, n_p)$ from all other parties, check that $|\mathsf{param}_i| = n_p$ for all j, and store the code array.

Public Data: On input $(\mathsf{PublicData}, a, d)$ from all parties, store d at address a in the data array.

Private Data: On input $(\mathsf{PrivateData}, a, d)$ from Party P_i and $(\mathsf{PrivateData}, i)$ from all other parties, store d at address a.

Run: On input (Run, pc) from all parties, execute the following:

1: **while** $pc \neq \perp$ **do**
2: Send (Tick) to the environment.
3: Load $(c_{pc}, \mathsf{param}_{pc})$ from the code.
4: $pc \leftarrow I_{c_{pc}}(\mathsf{param}_{pc})$
5: Send (Done) to the environment.

Reveal: On input (Reveal, i, a) from all parties, reveal the item at address a in the data array to Party P_i.

Abort: At any time, the environment can request aborting.

Bounds Check: For every access to the data or instruction array, if the index is within the bound, output $(\mathsf{WithinBounds})$ to the environment. Otherwise, output $(\mathsf{OutOfBounds})$ to all parties and the environment and abort.

Fig. 4. $\mathcal{F}_{machine}$

r_i read from memory if the instruction pattern mandates reading, and all but the last circuit also output a memory address a_i, a write flag f_i a value w_i to be written if the write flag is true and the instruction pattern mandates it.

Definition 1 (Instruction pattern). *An instruction pattern is a list* $\{(n_1, t_1), \ldots, (n_L, t_L)\}$ *where there is a state size* $n_i \in \mathbb{N}$ *and access type* $t_i \in \{\mathsf{Read}, \mathsf{Write}\}$ *for all* $i \in [L]$.

Definition 2 (Instruction). *An instruction with instruction pattern* $\{(n_1, t_1), \ldots, (n_L, t_L)\}$ *and parameter size* n_0 *is a list* $\{c_1, \ldots, c_{L+1}\}$ *of arithmetic circuits over* $\mathbb{F}$ *such that* c_i *has* $n_{i-1} + 2$ *inputs and* $n_i + 3$ *outputs. Let* $n_{L+1} = 1$. *The execution of an instruction is described in Algorithm 1.*

Algorithm 1 Instruction

Input: Parameters param, program counter pc
Output: Address of next instruction

$\mathsf{state}_0 \leftarrow \mathsf{param}$
$r_0 \leftarrow \perp$
for $i \in (1, \ldots, L)$ **do**
  Execute c_i with input $(r_{i-1}, pc, \mathsf{state}_{i-1})$ and output $(a_i, w_i, f_i, \mathsf{state}_i)$. ▷ $|\mathsf{state}_i| = n_i$ and $|\mathsf{state}_{i+1}| = n_{i+1}$
  if $t_i = \mathsf{Read}$ **then**
    $r_i \leftarrow$ the content of the data array at address a_i
  else
    $r_i \leftarrow \perp$
    if $f_i = 1$ **then**
      Write w_i to address a_i
return state_{L+1} ▷ $|\mathsf{state}_{L+1}| = 1$

As an example, we will now explain our implementation in Table 2 using the instruction $X_{X_i} \leftarrow X_j$. The first circuit c_0 gets as input the program counter pc and the instruction parameters (i, j, k) as the state, and it outputs the address i to be read. The content of this address is then input to c_1 as r_0 in the state. c_1 outputs the address j to be read and the state. The content of address j is stored as r_1 in the state. The circuit c_2 then requests r_1 to be stored at address r_0 (write flag 1). Finally, c_3 simply advances the program counter by one. The first two memory accesses only require reading the memory for every instruction (if at all), and the last one writing. Hence, $t_1 = \mathsf{Read}$, $t_2 = \mathsf{Read}$, and $t_3 = \mathsf{Write}$.

Theorem 2 (RAM model multiparty computation). *The oblivious machine allows the implementation of any RAM program in the model of Cook and Reckhow with the restriction that inputs are stored at the beginning of the execution and outputs are revealed at the end.*

Proof. Let $n_p = 3$ and $L = 3$. Table 2 shows the circuits to implement the first six instructions of Cook and Reckhow. Storing a constant, addition, and subtraction are done by using the relevant parameters (i, j, and k) as addresses and the constant, and indirect loading and storing by partially using previously loaded integers (r_1 and possibly r_0) as addresses. The TRA instruction mainly uses the fact that the output of c_L is used as address of the next instruction. Finally, reading inputs and revealing outputs can be done using PrivateInput/PublicInput and PrivateOutput/PublicOutput, respectively.

	$c_0(\bot, pc, (i,j,k))$	$c_1(r_0, pc, (i,j,k))$	$c_2(r_1, pc, (i,j,k,r_0))$	$c_3(\bot, pc, (i,j,k,r_0,r_1))$
$X_i \leftarrow C$	$(\bot,\bot,\bot,(i,j,k))$	$(\bot,\bot,\bot,(i,j,k,r_0))$	$(i, C, 1, (i,j,k,r_0,r_1))$	$(pc+1)$
$X_i \leftarrow X_j + X_k$	$(j,\bot,\bot,(i,j,k))$	$(k,\bot,\bot,(i,j,k,r_0))$	$(i, r_0+r_1, 1, (i,j,k,r_0,r_1))$	$(pc+1)$
$X_i \leftarrow X_j - X_k$	$(j,\bot,\bot,(i,j,k))$	$(k,\bot,\bot,(i,j,k,r_0))$	$(i, r_0-r_1, 1, (i,j,k,r_0,r_1))$	$(pc+1)$
$X_i \leftarrow X_{X_j}$	$(j,\bot,\bot,(i,j,k))$	$(r_0,\bot,\bot,(i,j,k,r_0))$	$(i, r_1, 1, (i,j,k,r_0,r_1))$	$(pc+1)$
$X_{X_i} \leftarrow X_j$	$(i,\bot,\bot,(i,j,k))$	$(j,\bot,\bot,(i,j,k,r_0))$	$(r_0, r_1, 1, (i,j,k,r_0,r_1))$	$(pc+1)$
TRA i if $X_j > 0$	$(j,\bot,\bot,(i,j,k))$	$(\bot,\bot,\bot,(i,j,k,r_0))$	$(\bot,\bot,0,(i,j,k,r_0,r_1))$	$(\mathsf{IfElse}(r_0 > 0, i, pc+1))$

(i, j, k): runtime arguments, r_0, r_1: values read from memory, pc: program counter

Table 2. Implementation of the random access machine by Cook and Reckhow

Private function evaluation. Recall that private function evaluation (PFE) allows two parties, one knowing a function and the other knowing some data, to compute the function on the data without revealing either. $\mathcal{F}_{machine}$ with private code input clearly facilitates this for functions formulated as a RAM-model computation. One party inputs the function using PrivateCode, and the other party inputs the data using PrivateData. The only leakage is the number of instructions computed. A malicious party can input a program that leaks some information through this number. However, we argue that this leakage is small compared to the amount of private data for practical scenarious because it only makes sense to use RAM model computation for larger data sets. Furthermore, if the parties reveal data at the end of the computation, a malicious party can input a program that leaks even more data there.

3.1 Implementation Using $\mathcal{F}_{ABBOA}$

We now propose protocol $\mathcal{P}_{machine}$ in Figure 5 as an implementation of the oblivious machine using MPC with oblivious arrays. At the core of our protocol lies the execution of all possible instruction in line 8. The oblivious selection can be done by multiplying the vector of results by a vector that contains 1 in one entry and 0 in the remaining entries. It is obvious that the protocol reveals the running time of the program. Note that the branching on t_i on line 9 does not reveal information because t_i is part of the instruction pattern, which is shared by all instructions.

Initialize: Initialize $\mathcal{F}_{ABBOA}$ for $\mathbb{F}$ and initialize an oblivious array $[D]$ of size n_d for the data.
Public Code: Initialize an oblivious array $[C]$ of size $n_c \cdot (1 + n_p)$ and fill it with $(c_0, \mathsf{param}_0), \ldots, (c_{n_c-1}, \mathsf{param}_{n_c-1})$.
Private Code: Initialize an oblivious array $[C]$ of size $n_c \cdot (1 + n_p)$ and fill it with $([c_0], [\mathsf{param}_0]), \ldots, ([c_{n_c-1}], [\mathsf{param}_{n_c-1}])$ input to the arithmetic black box $\mathcal{F}_{ABB}$ by Party P_i.
Public Data: Send $(\mathsf{WriteArray}([D], \mathcal{F}_{ABBOA}(\mathsf{PublicInput}, a), \mathcal{F}_{ABBOA}(\mathsf{PublicInput}, d), \mathcal{F}_{ABBOA}(\mathsf{PublicInput}, 1))$ to $\mathcal{F}_{ABBOA}$.
Private Data: Send $(\mathsf{WriteArray}([D], [a], [d], \mathcal{F}_{ABBOA}(\mathsf{PublicInput}, 1))$ to $\mathcal{F}_{ABBOA}$.
Run: The parties execute the following:

1: **while** $\mathsf{PublicOutput}([pc] \neq \perp)$ **do**
2: Load $([c_{pc}], [\mathsf{param}_{pc}])$ from the oblivious code array.
3: $[r_{-1}] \leftarrow \perp$
4: $[\mathsf{state}_{-1}] \leftarrow [\mathsf{param}]$
5: **for** $i = 0, \ldots, L$ **do**
6: **for** $j = 0, \ldots, n_I$ **do**
7: Execute c_i of instruction I_j on $([r_{i-1}], [pc], [\mathsf{state}_{i-1}])$ in $\mathcal{F}_{ABB}$ to get $([a_i^j], [w_i^j], [f_i^j], [\mathsf{state}_i^j])$.
8: Obliviously select the result $([a_i^{c_{pc}}], [w_i^{c_{pc}}], [f_i^{c_{pc}}], [\mathsf{state}_i^{c_{pc}}])$ as $([a_i], [w_i], [f_i], [\mathsf{state}_i])$.
9: **if** $t_i = \mathsf{Read}$ **then**
10: $[r_i] \leftarrow \mathcal{F}_{ABBOA}(\mathsf{ReadArray}, [a_i])$
11: **else**
12: $[r_i] \leftarrow \perp$
13: Send $(\mathsf{WriteArray}, [D], [a_i], [w_i], [f_i])$ to $\mathcal{F}_{ABBOA}$.
14: $pc \leftarrow \mathsf{state}_L$.

Reveal: Instruct $\mathcal{F}_{ABBOA}$ to open the a-th entry of the oblivious data array to Party P_i.
Bounds Check: Check every data and instruction access against the size of the respective array and reveal the result. Abort if the check fails.

Fig. 5. $\mathcal{P}_{machine}$

Theorem 3. *$\mathcal{P}_{machine}$ implements $\mathcal{F}_{machine}$ in the $\mathcal{F}_{ABBOA}$-hybrid model.*

Proof (Sketch). The power of the corrupted parties in the protocol is very limited because it only consists of calls to $\mathcal{F}_{ABBOA}$, which only reacts if all parties input the same information. Furthermore, $\mathcal{F}_{ABBOA}$ only reveals private data to any party in three situations. First, it does so so in the **Reveal** procedure, which corresponds to the same procedure of $\mathcal{F}_{machine}$. Second, the protocol reveals the result of comparison in line 1. Thirdly, it reveals if there is an access outside of array bounds. For these reasons, the simulator $\mathcal{S}_{machine}$ in Figure 6 is relatively simple. The first revelation can be simulated because $\mathcal{S}_{machine}$ learns the outputs of $\mathcal{F}_{machine}$ to corrupted parties. For the second revelation, it receives (Tick) whenever $\mathcal{F}_{machine}$ executes the loop body and (Done) after completion. Similarly, it receives all results of bounds checks. In the rest of the protocol, the adversary only learns handles, which $\mathcal{S}_{machine}$ can generate like $\mathcal{F}_{ABBOA}$ would. Hence, the view of the environment is indistinguishable.

Modeling the main loop with (Tick) and (Done) messages accounts for the fact that a program can run indefinitely. Otherwise, we would need to check whether a program halts on a given input, which is impossible for general programs due to the halting problem.

Complexity. The amount of communication and computation depends on the cost of accessing the oblivious array. Keller and Scholl [13] report an implementation based on Path ORAM [26] with access complexity in $O(\log^3 N)$ for arrays of length N. Using this, the complexity of initializing the data and the code array has cost in $O(n_c \log^3 n_c)$ and $O(n_d \log^3 n_d)$, respectively. The cost of running the machine is in $O(T(\log^3 n_c + L(\log^3 n_d + n_I))$ where T denotes the running time of the program and L denotes the number of memory

Generally: Emulate a copy of $\mathcal{F}_{ABBOA}$ by generating handles and aborting $\mathcal{F}_{machine}$ if the adversary demands so from $\mathcal{F}_{ABBOA}$.
Run: Whenever $\mathcal{F}_{machine}$ sends (Tick), simulate the protocol for another step revealing 0 in line 1. If $\mathcal{F}_{machine}$ sends (Done), reveal 1.
Reveal: If a corrupted party is due to receive data in the protocol, this is also the case in $\mathcal{F}_{machine}$. Simply forward it.
Bounds Check: Emulating $\mathcal{F}_{ABBOA}$, reveal the result according to (WithinBounds) or (OutOfBounds) received from $\mathcal{F}_{machine}$.

Fig. 6. $\mathcal{S}_{machine}$

accesses by instructions. In the following section, we will argue that $L = 3$ suffices to implement a machine that allows the compilation of arbitrary C code. Furthermore, we will show that $n_I = 10$ different instructions suffice to implement a small program matching a regular expression. All in all, our implementation features about 30 possible instructions.

4 Our Implementation

In this section, we will describe the details of our actual implementation. While the RAM model by Cook and Reckhow is powerful, more instructions are desirable to speed up computation. We begin by considering the differences between regular CPUs and our implementation.

Unlike in a regular CPU, there is no reason to have data registers because the memory has to be accessed in every step to maintain obliviousness. Therefore, all values are referred to by their addresses in the data array. The only register is the program counter referring the current instruction. Similarly, there is only one integer data type because having several types does not make sense in an oblivious execution. Because all possible instructions are executed at every step, any operation for a smaller integer type implies an operation for the larger type. Therefore, it is cheaper to execute the operation for the larger integer type only. This does not rule out the provision of floating point types. Since floating point operations are much more expensive than integer operations, and since they would have to be executed at every step, we did not implement floating point operations.

Our implementation only supports static memory allocation and no recursion due to the lack of stack pointer. However, there is no inherent reason for this limitation. The stack pointer could be implemented as another register. In such a scenario however, one has to define how to handle memory overflows. Making it public incurs the risk of leaking data while keeping it secret could lead to wasting time for a long computation on corrupted data.

Given the cost of ORAM accesses, the goal is to minimize the number thereof while still supporting all desired instructions. Three ORAM accesses, two reading and one writing, suffice for the kind of instructions that classical processors support. Every such instruction can be described as a four-tuple consisting of an identification of the instruction and three parameters, which can be an address or constant depending on the instruction. Not all parameters need to have a semantic meaning for an instruction.

We use the instruction pattern $(3, \mathsf{Read}), (4, \mathsf{Read}), (5, \mathsf{Write})$. The states are used to pass on the instructions parameters and the previously read values, that is, $\mathsf{state}_0 = \mathsf{param}$, $\mathsf{state}_1 = \mathsf{param} + (r_0)$, and $\mathsf{state}_2 = \mathsf{param} + (r_0, r_1)$. As in Table 2, all our instructions directly read the address specified by a parameter, which implies that c_0 simply outputs the relevant parameter as a_i.

Figure 7 lists all instructions used in the example in Figure 17. The example represents a matching algorithm for the regular expression "`ab*[cd]`" as explained in Section 5. To demonstrate our framework, we will now describe three instructions in detail.

We begin with "`add_const`", which could be implemented combining $X_i \leftarrow C$ and $X_i \leftarrow X_j + X_k$. However, we found that it is more efficient to have an extra instruction for adding a constant. As mentioned above, c_0 is the same for all instructions and causes the address at parameter `z` being read to r_0. c_1 only

Instruction	Description
`mov x y 0`	Copy the data at address `y` to address `x`.
`load x 0 z`	Copy the data at the address stored at address `z` to address `x`.
`store 0 y z`	Copy the data at address `z` to the address stored at address `y`.
`store_const x y 0`	Store the constant `y` at address `x`.
`eq_const x y z`	Compare the number at address `z` to the constant `y` and store 1 at address `x` in case of equality and 0 otherwise.
`add_const x y z`	Add the number stored at address `z` and the constant `y`, and store the result at address `x`.
`ult_pos_const x y z`	Test if the unsigned number at address `z` is less than the positive constant `y` and store 1 at address `x` if yes and 0 otherwise.
`ule_pos_const x y z`	Test if the unsigned number at address `z` is less or equal than the positive constant `y` and store 1 at address `x` if yes and 0 otherwise.
`jmp x 0 0`	Jump to the instruction at address `x`.
`jmp_ind 0 0 z`	Jump to the instruction at the address stored at `x`.
`br x y z`	Jump to the instruction at address `x` if the number at address `z` is 1 and to the instruction at address `y` if the number is 0. Undefined behaviour in any other case.

Fig. 7. Instructions used in Figures 10 and 17.

forwards the parameters and r_0, and c_2 causes $r_0 + \texttt{y}$ to be stored at address `x`. c_3 causes the program to continue with the next instruction.

The "`load`" instruction corresponds to $X_i \leftarrow X_{X_j}$ in Table 2. Again, c_0 causes the address at parameter `z` being read to r_0. c_1 then enables the dereferencing by causing the address r_0 being read to r_1, and c_2 causes r_1 to being written at address `x`. Finally, c_3 causes the execution to continue with the next instruction in the code memory.

Finally, the "`br`" instruction is similar to the TRA instruction in Table 2. It again starts with c_0 causing the address `z` being read to r_0. c_1 and c_2 do nothing but forward the parameters and r_0. c_3 then causes the execution to jump to either `x` or `y` in the code memory, depending on r_0. IfElse denotes the selection circuit $\mathsf{IfElse}(c, a, b) = c \cdot (b - a) + a$.

It is easy to see that our framework allows for most instructions one expects in a CPU because they fall in one of three categories represented by the above descriptions. Binary operations can be implemented similar to `add_const` by loading the operands and storing the result, indirect memory are similar to `load`, and conditional and unconditional jumps can be specified similarly to `br`.

4.1 Compilation

To compile C code, we use Clang from the LLVM project [27]. The LLVM project provides a modular compiler toolchain. Clang parses C code and can compile it to the LLVM internal representation. This representation consists of CPU-like instructions for an abstract CPU with infinitely many registers.

We use a Python script to compile this internal representation for our machine. This allows to compile simple C programs without having to write an LLVM specification of our machine.

Since our machine does not support registers, the compiler has to allocate memory space for every register. For simplicity, every register is put in a separate space in memory. One could use static analysis to reduce the amount of space used.

We will now present a few examples of the compilation process. The first example is a "for" loop populating an array. Figure 8 shows the C code. Compiled to LLVM intermediate representation (Figure 9), the code is divided in five basic blocks: the code before the loop, the loop condition check, the loop body, the loop increment, and the code after the loop. Shown in Figure 10, the code for our machine contains less instructions than the LLVM code because the compiler gets rid of unnecessary instructions, such as jumping to the next instruction in the code and loading from memory to register. For the latter case, note that our machine does not support registers. The loop variable `i` is stored in position 8 in the memory, initialized to 0 and compared to the constant 5 in instruction 2. The result of this comparison goes to position 9 in

memory, which is used by the branching instruction in instruction 3. Furthermore, memory position 10 holds the address of the array element accessed in the loop body. The array starts at position 3; hence, instruction 4 adds 3 to the loop variable to determine the address of the array element. The loop variable is then stored in the address by instruction 5. For incrementing, instruction 6 adds 1 to the loop variable and stores the result in position 11, which is then copied to position 8 by instruction 7. The execution then jumps back to the condition check in instruction 2.

```
1 int main() {
2   unsigned long a[5];
3   for (unsigned long i = 0; i < 5; i++)
4     a[i] = i;
5 }
```

Fig. 8. A "for" loop populating an array in C.

5 Efficient Private Regular Expression Matching with Minimal Leakage

Consider the problem of two parties wanting to decide whether a string known by one party matches a regular expression known by the other. A regular expression can be modelled by a finite state machine, which in turn can be implemented in C using mainly switch and goto statements. Bumbulis and Cowan [3] provide an implementation of such a compilation. Therefore, the oblivious machine solves the problem by the party holding the regular expression inputting an appropriate program using PrivateCode. In the full version, we will show an example of the resulting C, LLVM, and machine code.

5.1 Complexity

Using Path ORAM for the oblivious arrays, loading the code and the input string takes time in $O(n \log^3 n)$ in both cases with n denoting the code size and the input size, respectively. For the regular expression, this means quasi-linear time in the size of the finite state machine. The main execution then takes time in $O(T(\log^3 n_c + \log^3 n_d))$ for T, n_c, and n_d denoting the running time of the machine, the code size, and the size of the input string, respectively. The running time is dominated the by the comparisons made by the C code. This depends both on the input data and the ordering of the comparison within a switch statement. However, T can be upper bounded by $O(n_d n_m)$ where n_m denotes the maximum number of comparisons in a single switch statement, which in turn is less than the number of symbols. From a certain size of switch statements, it is more efficient to use branch tables instead of consecutive comparisons. In this case, the loading complexity of the particular switch statement is in $O(n_s)$ for n_s being the number of symbols and the execution complexity is constant. We did not follow this avenue in our implementation, but point out that it would possible using the `jmp_ind` instruction.

5.2 Security

The oblivious machine leaks the running time T, which is linear in the number of the comparison performed and the size of the input string. The latter is public by the fact that the usage of the oblivious machine reveals the input size. However, the former depends on the regular expression, its precise compilation, and the input string. The party holding the regular expression could modify the switch statements such that they have constant size. However, this can come at considerable cost if they switch statements highly vary in size.

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca [5 x i64], align 16
  %i = alloca i64, align 8
  store i32 0, i32* %retval
  store i64 0, i64* %i, align 8
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i64* %i, align 8
  %cmp = icmp ult i64 %0, 5
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %1 = load i64* %i, align 8
  %2 = load i64* %i, align 8
  %arrayidx = getelementptr inbounds [5 x i64]* %a, i32 0, i64 %2
  store i64 %1, i64* %arrayidx, align 8
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %3 = load i64* %i, align 8
  %inc = add i64 %3, 1
  store i64 %inc, i64* %i, align 8
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %4 = load i32* %retval
  ret i32 %4
}
```

Fig. 9. A "for" loop populating an array in the LLVM intermediate representation.

```
# main()
  # entry:
    store_const 2 0 0 # 0
    store_const 8 0 0 # 1
  # for.cond:
    ult_pos_const 9 5 8 # 2
    br 4 9 9 # 3
  # for.body:
    add_const 10 3 8 # 4
    store 0 8 10 # 5
  # for.inc:
    add_const 11 1 8 # 6
    mov 8 11 0 # 7
    jmp 2 0 0 # 8
  # for.end:
    mov 0 2 0 # 9
    jmp 11 0 0 # 10
```

Fig. 10. A "for" loop populating an array in our machine code.

6 Experiments

In order to benchmark our construction, we have implemented it based on the so-called SPDZ scheme by Damgård et al. [8], which provides active security against an adversary corrupting all but one party. It works in the preprocessing model, that is, there is a data-independent offline phase in which correlated randomness is generated. In the case of SPDZ, the offline phase uses homomorphic encryption to generate secret sharings of random multiplicative triples (a, b, ab) in a finite field with some authentication. The online phase then uses the correlated randomness to compute the product of actual inputs.

In our implementation, we use 64-bit integers as subset of $\mathbb{F}_p$ for a 128-bit prime p. The gap is necessary to accommodate for the statistically secure bit decomposition protocol with security parameter 40.

Figure 11 shows the online phase clock rate for a minimal program that is executed with varying sizes of the data memory. The offline phase is about in magnitude of a 1000 times slower. However, it is highly parallelizable, that is, it can distributed among several machines, which does not hold for the online phase. Furthermore, this cost is due to providing active security, which similar works do not offer.

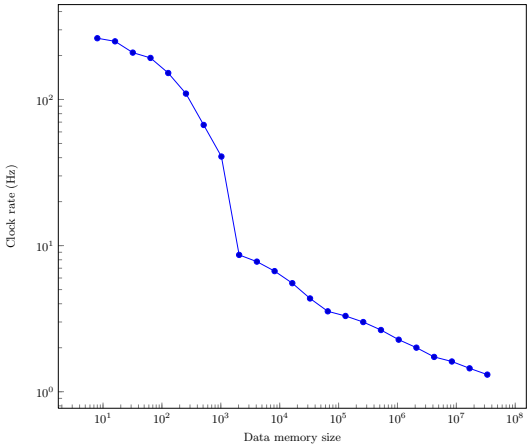

Fig. 11. Online phase clock rate of the machine.

6.1 Comparison with Purely Using Oblivious Arrays

In order to compare the performance of the oblivious machine with programs implemented using oblivious arrays directly, we have benchmarked Dijkstra's algorithm using our toolchain. The results in Figure 12 suggest that using the oblivious machine instead of the implementation by Keller and Scholl is about 100 times slower. However, the comparison is not entirely fair because the previous implementation leaks the algorithm being computed whereas the oblivious machine does not. In other words, the price to pay to hide the computation is a factor of 100 in this case.

6.2 Regular Expression Matching

We have implemented our protocol for regular expression matching for a string of length 1024 and randomly generated regular expressions of varying complexity. Figure 13 shows our results. We found that the total number of transitions is the most appropriate measure for the complexity of a finite state machine. This coincides with the number of comparisons in the machine code. At more than ten thousand transitions, loading the code becomes the dominant part of the computation. However, if several strings are to be matched to the same regular expression, this is a one-off cost. Table 3 shows the smaller expressions we used for our experiments.

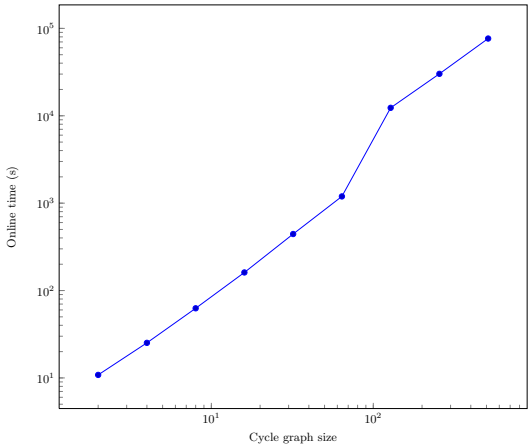

Fig. 12. Dijkstra's algorithm on cycle graphs.

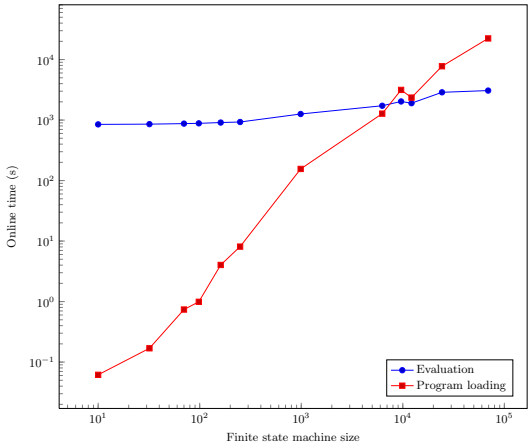

Fig. 13. Regular expression matching online time.

FSM Size	Regular Expression
10	`[ZrqupR]`
32	`[aYoNPCI7O][Lxdo][3jHl7]`
70	`(([x2YUux]|[FEb6o])([ssUWGaGuD]?)?)[nO][LwhCA][YOrp6xc]WkaNjg5M`
98	`[P]([zOxwIv48]+)([IEm][^isgQn4B]*)`
161	`([M]|(XP[t]*))[slUW8XiVe][iTS2Y86E][ykSh9uE][fAu]9TOg(Umks(do([^t]|[PEv3e5])+)e62e[iIl]*)`

Table 3. Regular expressions used in our experiments.

7 Conclusion and Future Directions

We have presented a theoretical model for multiparty RAM computation, a concrete protocol, and an implementation as well as an application in the form of private regular expression matching. As future direction we suggest research into quantifying the leakage by the running time of a RAM program. This would allow to navigate the trade-off between the fastest execution of a program with leakage and the overhead by adding padding operations to programs in order to hide the number of comparisons in our regular expression matching scheme for example.

Our experiments have shown that the oblivious machine runs at a few Hertz for larger data memory size, which is about a billion times slower than a regular CPU. Obviously, one cannot hope to achieve a similar speed, but recent ORAM schemes optimized for circuit implementation should allow to improve at least one or two orders of magnitude. Another issue is the round complexity of secret-sharing-based MPC schemes, which is linear in the circuit rounds. Analyzing our implementation, we come to the conclusion that this is the bottleneck. Two-party computation based on Yao's garbled circuits does not suffer from this because it has constant rounds. However, implementations using garbled circuits do not necessarily beat the ones secret sharing [6]. It remains to be seen which approach is more efficient.

Acknowledgments

We would like to thank Peter Scholl for various comments and suggestions. This work has been supported in part by EPSRC via grant EP/I03126X.

References

1. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Simon [25], pages 1–10.
2. E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.
3. P. Bumbulis and D. D. Cowan. RE2C - a more versatile scanner generator. *ACM Lett. Program. Lang. Syst*, 2:70–84, 1994.
4. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In Simon [25], pages 11–19.
5. S. A. Cook and R. A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.
6. I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In I. Visconti and R. D. Prisco, editors, *Security and Cryptography for Networks - 8th International Conference, SCN 2012, Amalfi, Italy, September 5-7, 2012. Proceedings*, volume 7485 of *Lecture Notes in Computer Science*, pages 241–263. Springer, 2012.
7. I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography*, pages 144–163. Springer, 2011.
8. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.
9. C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with HE-over-ORAM architecture. Cryptology ePrint Archive, Report 2014/345, 2014. `http://eprint.iacr.org/2014/345`.
10. C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled RAM revisited. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2014.
11. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In A. V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

12. S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 513–524. ACM, 2012.
13. M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In Sarkar and Iwata [23], pages 506–525.
14. F. Kerschbaum. Practical private regular expression matching. In S. Fischer-Hübner, K. Rannenberg, L. Yngström, and S. Lindskog, editors, *Security and Privacy in Dynamic Environments*, volume 201 of *IFIP International Federation for Information Processing*, pages 461–470. Springer US, 2006.
15. J. Launchbury, D. Archer, T. DuBuisson, and E. Mertens. Application-scale secure multiparty computation. In Z. Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 8–26. Springer Berlin Heidelberg, 2014.
16. Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.
17. C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient RAM-model secure computation. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 623–638, Washington, DC, USA, 2014. IEEE Computer Society.
18. C. Liu, Y. Huang, E. Shi, J. Katz, and M. W. Hicks. Automating efficient ram-model secure computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 623–638. IEEE Computer Society, 2014.
19. C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376. IEEE Computer Society, 2015.
20. P. Mohassel, S. S. Sadeghian, and N. P. Smart. Actively secure private function evaluation. In Sarkar and Iwata [23], pages 486–505.
21. R. Ostrovsky and V. Shoup. Private information storage. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 294–303. ACM, 1997.
22. N. Pippenger and M. J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.
23. P. Sarkar and T. Iwata, editors. *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, volume 8874 of *Lecture Notes in Computer Science*. Springer, 2014.
24. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer, 2011.
25. J. Simon, editor. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. ACM, 1988.
26. E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM Conference on Computer and Communications Security*, pages 299–310. ACM, 2013.
27. The LLVM Project. clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.
28. X. S. Wang, S. D. Gordon, A. McIntosh, and J. Katz. Secure computation of mips machine code. Cryptology ePrint Archive, Report 2015/547, 2015. `http://eprint.iacr.org/2015/547`.
29. X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi. SCORAM: oblivious RAM for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 191–202, 2014.

A Regular Expression Example

For the regular expression `ab*[de]`, Figures 14 to 17 show its implementation as C code, LLVM intermediate representation, and instruction code for the oblivious machine.

```
1  long match(char* YYCURSOR) {
2    char* YYMARKER;
3
4    while (1)
5    {
6      char yych;
7
8      yych = *YYCURSOR;
9      switch (yych) {
10     case 'a':  goto yy2;
11     default:  goto yy5;
12     }
13 yy2:
14     ++YYCURSOR;
15     yych = *YYCURSOR;
16     switch (yych) {
17     case 'b':  goto yy2;
18     case 'c':
19     case 'd':  goto yy7;
20     default:  goto yy4;
21     }
22 yy4:
23 yy5:
24     ++YYCURSOR;
25     { return 0; }
26 yy7:
27     ++YYCURSOR;
28     { return 1; }
29   }
30 }
```

Fig. 14. The regular expression "`ab*[de]`" compiled to C

```
define i64 @match(i64* %YYCURSOR) #0 {
entry:
  %retval = alloca i64, align 8
  %YYCURSOR.addr = alloca i64*, align 8
  %YYMARKER = alloca i64*, align 8
  %yych = alloca i64, align 8
  store i64* %YYCURSOR, i64** %YYCURSOR.addr, align 8
  br label %while.body

while.body:                                       ; preds = %entry
  %0 = load i64** %YYCURSOR.addr, align 8
  %1 = load i64* %0, align 8
  store i64 %1, i64* %yych, align 8
  %2 = load i64* %yych, align 8
  br label %LeafBlock

LeafBlock:                                        ; preds = %while.body
  %SwitchLeaf = icmp eq i64 %2, 97
  br i1 %SwitchLeaf, label %sw.bb, label %NewDefault

sw.bb:                                            ; preds = %LeafBlock
  br label %yy2

NewDefault:                                       ; preds = %LeafBlock
  br label %sw.default

sw.default:                                       ; preds = %NewDefault
  br label %yy5

yy2:                                              ; preds = %sw.bb1, %sw.bb
  %3 = load i64** %YYCURSOR.addr, align 8
  %incdec.ptr = getelementptr inbounds i64* %3, i32 1
  store i64* %incdec.ptr, i64** %YYCURSOR.addr, align 8
  %4 = load i64** %YYCURSOR.addr, align 8
  %5 = load i64* %4, align 8
  store i64 %5, i64* %yych, align 8
  %6 = load i64* %yych, align 8
  br label %NodeBlock
```

Fig. 15. The regular expression "`ab*[de]`" compiled to LLVM code (part one)

```
NodeBlock:                                        ; preds = %yy2
  %Pivot = icmp ult i64 %6, 99
  br i1 %Pivot, label %LeafBlock2, label %LeafBlock4

LeafBlock4:                                       ; preds = %NodeBlock
  %.off = add i64 %6, -99
  %SwitchLeaf5 = icmp ule i64 %.off, 1
  br i1 %SwitchLeaf5, label %sw.bb2, label %NewDefault1

LeafBlock2:                                       ; preds = %NodeBlock
  %SwitchLeaf3 = icmp eq i64 %6, 98
  br i1 %SwitchLeaf3, label %sw.bb1, label %NewDefault1

sw.bb1:                                           ; preds = %LeafBlock2
  br label %yy2

sw.bb2:                                           ; preds = %LeafBlock4
  br label %yy7

NewDefault1:                                      ; preds = %LeafBlock2, ↩
    %LeafBlock4
  br label %sw.default3

sw.default3:                                      ; preds = %NewDefault1
  br label %yy4

yy4:                                              ; preds = %sw.default3
  br label %yy5

yy5:                                              ; preds = %yy4, %sw.↩
    default
  %7 = load i64** %YYCURSOR.addr, align 8
  %incdec.ptr4 = getelementptr inbounds i64* %7, i32 1
  store i64* %incdec.ptr4, i64** %YYCURSOR.addr, align 8
  store i64 0, i64* %retval
  br label %return

yy7:                                              ; preds = %sw.bb2
  %8 = load i64** %YYCURSOR.addr, align 8
  %incdec.ptr5 = getelementptr inbounds i64* %8, i32 1
  store i64* %incdec.ptr5, i64** %YYCURSOR.addr, align 8
  store i64 1, i64* %retval
  br label %return

return:                                           ; preds = %yy7, %yy5
  %9 = load i64* %retval
  ret i64 %9
}
```

Fig. 16. The regular expression "`ab*[de]`" compiled to LLVM code (part two)

```
# match()
  # entry:
    mov 1028 1026 0 # 0
  # while.body:
    load 1031 0 1028 # 1
    mov 1030 1031 0 # 2
  # LeafBlock:
    eq_const 1032 97 1030 # 3
    br 5 16 1032 # 4
  # yy2:
    add_const 1033 1 1028 # 5
    mov 1028 1033 0 # 6
    load 1034 0 1028 # 7
    mov 1030 1034 0 # 8
  # NodeBlock:
    ult_pos_const 1035 99 1030 # 9
    br 14 11 1035 # 10
  # LeafBlock4:
    add_const 1036 -99 1030 # 11
    ule_pos_const 1037 1 1036 # 12
    br 20 16 1037 # 13
  # LeafBlock2:
    eq_const 1038 98 1030 # 14
    br 5 16 1038 # 15
  # yy5:
    add_const 1039 1 1028 # 16
    mov 1028 1039 0 # 17
    store_const 1027 0 0 # 18
    jmp 23 0 0 # 19
  # yy7:
    add_const 1040 1 1028 # 20
    mov 1028 1040 0 # 21
    store_const 1027 1 0 # 22
  # return:
    mov 1024 1027 0 # 23
    jmp_ind 0 0 1025 # 24
```

Fig. 17. The regular expression "`ab*[de]`" compiled for the oblivious machine