

# The Oblivious Machine

## or: How to Put the C into MPC

Marcel Keller

Department of Computer Science, University of Bristol  
mks.keller@gmail.com

**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 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 in 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 [21] 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 Appendix A.

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 [25], 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. [27], practical instantiations of oblivious data structures for multiparty computation have been proposed, both for Yao’s garbled circuits [12, 22, 33] and secret-sharing based MPC [16]. 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, 22] have introduced the concept of secure RAM-model computation, their notions remain rather abstract. Furthermore, Liu et al. [22] 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 [16] 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. Instead, we execute all possible instructions in every computation step and then obliviously select the desired result. Since the number of possible instructions is relatively low, the most efficient oblivious selection simply computes the inner product with a vector that contains 1 in one entry and 0 otherwise.

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 [17] 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. [19] 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. [24]. 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 [16], 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. As an example, they propose a secure computation of Dijkstra’s algorithm involving a manual rewrite of the algorithm. In contrast, our scheme allows to execute any algorithm written in C.

Similarly, SCVM [22] and OblivM [23] 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 described in Section 4. Furthermore, both SCM and OblivM do not hide the program being executed.

Wang et al. [33] 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 an independent preprint [32], Wang et al. propose a compiler of bytecode for a particular processor (MIPS) to garbled circuits. While their independent work shares some characteristics with ours, there are differences in various 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 computations, this is limited to relatively small programs because it requires the computation of every possible execution path of the program. Furthermore, this inhibits private function evaluation. In comparison, our approach works for every program because it allows memory accesses in every step.

Secondly, Wang et al.’s 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 100 times more expensive than the online phase using the recent improvement [15] to the offline phase of SPDZ.

Songhori et al. [28] propose TinyGarble, a framework for garbled circuit computation. While they claim to provide the first scalable emulation of a general purpose processor, they explicitly do not support ORAM, which incurs a linear overhead in contrast to the polylogarithmic overhead of our construction. Furthermore, the examples they present (sum, Hamming distance, comparison, multiplication, matrix multiplication, AES, and SHA3) do not require branching. Similarly, there are several works presenting compilers from high-level languages to two- or multiparty computation [13, 14, 18, 34, 35], none of which consider proper branching on secret values. While the authors of Obliv-C [34] consider ORAM, they present examples (edit distance computation and millionaire’s problem) that do not require ORAM.

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 [26] 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 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 considerably from multiparty computation. In the former, a server can compute a RAM program using a regular CPU and then has to prove to prove this correctness of the computation. In the setting of MPC, we are concerned with the computation of the RAM program itself.

## 2 From Oblivious Arrays to Oblivious Computation

In this section, we summarize Appendix B.

Keller and Scholl [16] present an implementation of oblivious arrays for any MPC scheme that provides an arithmetic black box while the security properties of the underlying scheme remain intact. An oblivious array in the context of MPC is an array that can be accessed by a secret index without revealing it. Oblivious RAM allows to implement an oblivious array with polylogarithmic access cost.

Even with the result above, all known basic MPC schemes are data-oblivious but leak which operation is executed. This can be avoided by executing a set of operations in every step of the computation and obliviously selecting the actual result. Storing this private selectors (or instructions) in an oblivious array and having a secret program counter point at the current one, we get the oblivious equivalent of a random access machine because the secret program counter can be changed obliviously as well. We call this the oblivious machine. It allows branching (e.g., “if” statements) on secret data without revealing it except the leakage through the total computation time.

Furthermore, the properties of the oblivious machine allow to implement private function evaluation, where the code is only known to one party. This is possible because it is straightforward to initialize the oblivious array holding the instructions with secret inputs from one party.

**Complexity.** The amount of communication and computation depends on the cost of accessing the oblivious array. Keller and Scholl [16] report an implementation based on Path ORAM [29] with access complexity in  $O(\log^3 N)$  for arrays of length  $N$ . Using this, the complexity of initializing the data and the code arrays has cost in  $O(n \log^3 n)$ , where  $n$  denotes the maximal size of the data and code array. Similarly, the cost of running the machine is in  $O(T \log^3 n)$  where  $T$  denotes the running time of the program if the number of possible instructions is constant.

We have also implemented Circuit ORAM [31], but we found it to be slower in our context than Path ORAM, despite the improved asymptotic complexity. We assume that this is due to the higher number of rounds of multiplications in Circuit ORAM, which translate to communication rounds in secret sharing-based MPC.

## 3 Our Implementation

In this section, we will describe the details of our implementation. 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. In other words, if the memory would not be accessed in a particular step, the adversary would learn some information about the current program counter by ruling out all positions in the program that do not involve a memory access. Over time, the adversary could learn the exact position in the program, which in turn would rule out branching on secret variables. Wang et al. [32] use static analysis to evaluate at which execution no possible branch accesses the memory to avoid accessing in every step. However, this approach prevents private function evaluation, and the cost analysis grows exponentially in the number of branches in the worst case.

Lacking registers, 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.

We found it more practical to implement a Harvard-like architecture, where the memory for instructions is separated from the data storage, instead of the common Von Neumann architecture, where the two are

mixed. First, there is an efficiency gain in ORAM by storing uniform tuples under every address (and the structure of the stored information differs between the program and data memory), and second, the overhead of ORAM implies that two smaller oblivious arrays are cheaper to access than one combined array.

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.

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

**Fig. 1.** Instructions used in Figures 4 and 17.

Figure 1 lists all instructions used in the example in Figures 4 and 17. The example represents a matching algorithm for the regular expression “`ab*[cd]`” as explained in Section 4.

Before, after, and between ORAM accesses, all computation is purely in the form of circuits, obviously selected as described in the Appendix B. The circuits not only compute arithmetic operations, they also compute the addresses to be read from the data array and the next program counter of the next instruction.

For example, in `add_const`, one circuit makes sure that `z` is read from the memory, one circuit computes the addition with `y`, one circuit outputs `x` as the writing address, and the final circuit increments the program counter by one. Similarly, most circuits for `load` simply redirect addresses and the value read from memory. In addition to that, the last circuit of `br` set the program counter to  $y + x \cdot (z - y)$  where `x`, `y`, and `z` denote the values at addresses `x`, `y`, and `z`.

It is easy to see that all instructions fall in one of the 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`.

### 3.1 Compilation

To compile C code, we use Clang from the LLVM project [30]. 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. While the latter might be more elegant, our method seemed more time-efficient considering that our machine shares few characteristics with real CPUs that allow optimizations, e.g., registers.

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 have implemented and tested the following features of C:

- Integer addition, subtraction, multiplication, comparison, and shifting
- Pointers
- Arrays (also multi-dimensional)
- Structs
- `for` and `while` loops
- `switch` statements
- `goto` statements
- Functions (without recursion)

On the other hand, we have left out the following features:

- Floating-point operations
- Recursive function calls
- Dynamic memory allocation

While the omitted features clearly are important for a complete support of C, we argue that the main restrictions of circuit-based computation (branching and array accesses) are entirely overcome by our implementation. Furthermore, note that the missing features are only restricted in our implementation and not by our theoretical model presented in Appendix B.

We will now present a few examples of the compilation process. The first example is a “for” loop populating an array. Figure 2 shows the C code. Compiled to LLVM intermediate representation (Figure 3), 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 4, 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.

Another example can be found in Appendix C.

## 4 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 modeled by a finite state machine,

```

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

```

**Fig. 2.** A “for” loop populating an array in C.

```

1 define i32 @main() #0 {
2   entry:
3     %retval = alloca i32, align 4
4     %a = alloca [5 x i64], align 16
5     %i = alloca i64, align 8
6     store i32 0, i32* %retval
7     store i64 0, i64* %i, align 8
8     br label %for.cond
9
10  for.cond:                                ; preds = %for.inc, %entry
11     %0 = load i64* %i, align 8
12     %cmp = icmp ult i64 %0, 5
13     br i1 %cmp, label %for.body, label %for.end
14
15  for.body:                                ; preds = %for.cond
16     %1 = load i64* %i, align 8
17     %2 = load i64* %i, align 8
18     %arrayidx = getelementptr inbounds [5 x i64]* %a, i32 0, i64 %2
19     store i64 %1, i64* %arrayidx, align 8
20     br label %for.inc
21
22  for.inc:                                  ; preds = %for.body
23     %3 = load i64* %i, align 8
24     %inc = add i64 %3, 1
25     store i64 %inc, i64* %i, align 8
26     br label %for.cond
27
28  for.end:                                  ; preds = %for.cond
29     %4 = load i32* %retval
30     ret i32 %4
31 }

```

**Fig. 3.** A “for” loop populating an array in the LLVM intermediate representation.

```

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

```

Fig. 4. A “for” loop populating an array in our machine code.

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 securely. In Appendix C, we show an example of the resulting C, LLVM, and machine code.

#### 4.1 Complexity

Using Path ORAM for the oblivious arrays, loading the code and the input string takes time in  $O(n \log^3 n)$  with  $n$  denoting the maximum of the code size and input size. 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 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. For example, if it makes a difference whether the next character is ‘a’, ‘b’, or anything else, one can first check for ‘a’ or first check for ‘b’. The order of the checks influences the number of checks computed if the next character is ‘a’ or ‘b’. 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. For example, if every character of the alphabet is treated differently, one would have to conduct 26 comparisons in order to jump to the right position in the code. A branch table is a list of jump instructions where the first one jumps to the code for ‘a’, the second one jumps to the code for ‘b’ etc. To use this table, the preceding code simply computes a jump address by adding the correct base address to the number representing the character and then jumps to the result. Of course the code still needs to conduct two comparisons to check whether the next character is within the bounds of the branch table. 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 be possible using the `jmp_ind` instruction.

#### 4.2 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.<sup>1</sup> 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. This could for example be achieved using the approach with branch tables explained above.

## 5 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 online phase requires secret sharings of random multiplicative triples  $(a, b, ab)$  in a finite field with some authentication to compute the product of actual inputs.

The original SPDZ protocol used somewhat homomorphic encryption to generate these multiplication triples, but Keller et al. [15] recently found that it is more efficient to use oblivious transfer for the offline phase instead. Therefore, we use their timings in our figures.

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. Note that this refers to a  $2^{-40}$  comparison advantage in the security proof, not the possibility of breaking the protocol with  $2^{40}$  complexity.

All experiments were conducted on two off-the-shelf machines with Intel Core i7 processors connected by a 1 GBit/s local network.

Figure 5 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 100 times slower. However, it is highly parallelizable, that is, it can be 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. Note that the “bump” at memory size 1000 stems from the fact that for smaller sizes it is more efficient to use linear scanning instead of the more intricate Path ORAM.

Per clock cycle, the complexity ranges from 11 KBytes sent in 40 rounds for memory size 2 to 7.6 MBytes in 2739 rounds for memory size  $2^{24}$ .

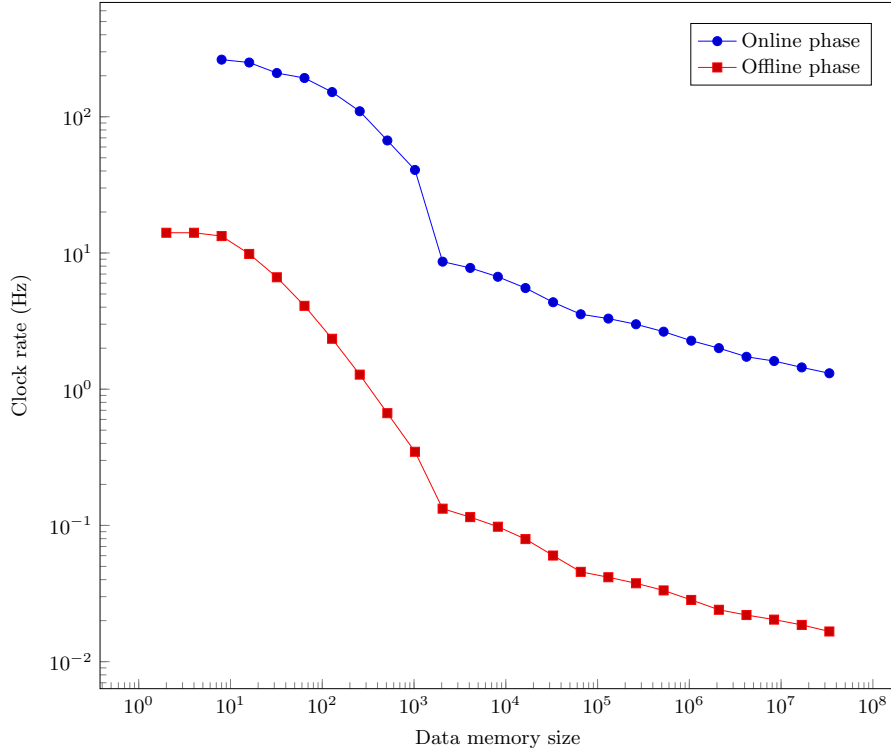
### 5.1 Comparison to Non-Oblivious Computation

It is clear that our approach is slower than plain multiparty computation for programs that can be efficiently computed as circuits. 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 6 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. Furthermore, they have to rewrite the algorithm because its structure with nested variable length loops does not lend itself to oblivious computation. In other words, the price to pay to hide the computation or to avoid manual rewriting is a factor of 100 in this case. We consider this to be representative for algorithms that require the use of oblivious arrays for an efficient implementation. Again, the “bump” in the figure is explained by the change in the implementation of the oblivious array.

### 5.2 Comparison with Wang et al.’s Secure MIPS Computation

We have compiled and run the set intersection example by Wang et al. [32]. Table 1 shows that our running times are comparable to the unoptimized times by Wang et al. Their optimization involves static analysis of the program, and thus inhibits private function evaluation. Furthermore, note that their implementation uses semi-honest garbled circuits while ours provides active security.

<sup>1</sup> While there exist works on size-hiding secure computation [20], we do not think that those approaches are compatible with ours. In particular, they seem to be “one-shot” while ORAM requires continuous computation.



**Fig. 5.** Clock rate of the machine.

Input size per party	64 inputs	256 inputs	1024 inputs
Wang et al. baseline	58.35 s	324.09 s	3068.19 s
Wang et al. optimized	2.77 s	12.96 s	108.45 s
This work	6.43 s	44.12 s	1346.82 s

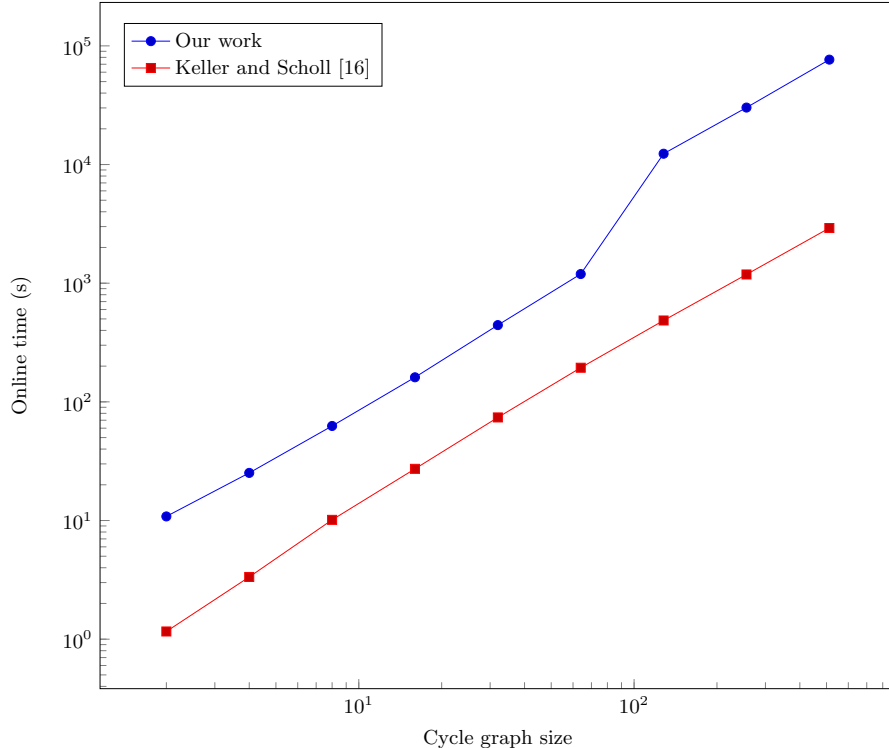
**Table 1.** Set intersection.

### 5.3 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 7 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 2 shows the smaller expressions we used for our experiments. We used randomly generated expressions, which explains the irregular increase in the size of the corresponding finite state machines.

## 6 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



**Fig. 6.** Dijkstra’s algorithm on cycle graphs.

adding padding operations to programs in order to hide the number of comparisons in our regular expression matching scheme for example.

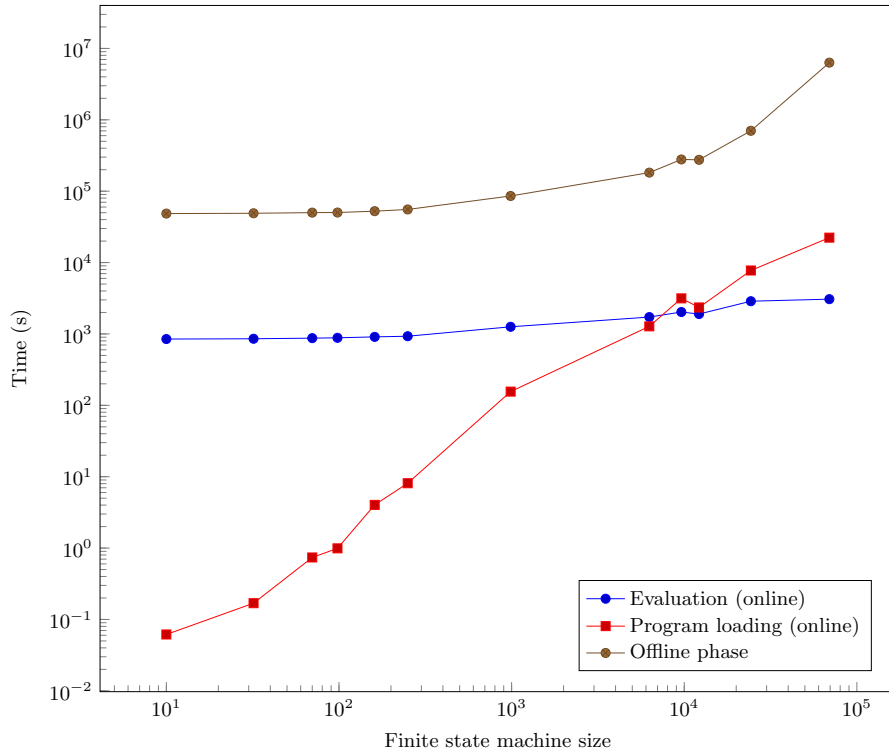
Our experiments have shown that the oblivious machine runs at a few Hertz for larger data memory size, which is about a billion times slower than a regular CPU. Obviously, one cannot hope to achieve a similar speed, but recent ORAM schemes optimized for circuit implementation should allow to improve 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 grants EP/I03126X, EP/M012824, and EP/N021940/1.

## References

1. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.
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, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, Aug. 2013.



**Fig. 7.** Regular expression matching timings.

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 (abstract) (informal contribution). In C. Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, page 462. Springer, Heidelberg, Aug. 1988.
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, *SCN 12*, volume 7485 of *LNCS*, pages 241–263. Springer, Heidelberg, Sept. 2012.
7. I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In Y. Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 144–163. Springer, Heidelberg, Mar. 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, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, Aug. 2012.
9. C. Gentry, S. Halevi, C. S. Jutla, and M. Raykova. Private database access with HE-over-ORAM architecture. In T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 172–191. Springer, Heidelberg, June 2015.
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 2014*, volume 8441 of *LNCS*, pages 405–422. Springer, Heidelberg, May 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. Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 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 CCS 12*, pages 513–524. ACM Press, Oct. 2012.

FSM Size	Regular Expression
10	[ZrqupR]
32	[aYoNPCi70] [Lxdo] [3jH17]
70	(( [x2YUux]   [FEb6o] ) ( [ssUWGaGuD] ? ) ? ) [n0] [LwhCA] [Y0rp6xc] WkaNjg5 M
98	[P] ( [z0xwIv48] + ) ( [IEm] [ ^ i s g Q n 4 B ] * )
161	( [M]   ( XP [t] * ) ) [s1UW8XiVe] [iTS2Y 86E] [ykSh9uE] [fAu] 9T0g ( Umks ( do ( [ ^ t ]   [PEv3e5] ) + ) e62e [i1l] * )

**Table 2.** Regular expressions used in our experiments.

13. W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *ACM CCS 10*, pages 451–462. ACM Press, Oct. 2010.
14. A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 12*, pages 772–783. ACM Press, Oct. 2012.
15. M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 16*, pages 830–842. ACM Press, Oct. 2016.
16. M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 506–525. Springer, Heidelberg, Dec. 2014.
17. F. Kerschbaum. Practical private regular expression matching. In *Security and Privacy in Dynamic Environments*, volume 201 of *IFIP International Federation for Information Processing*, pages 461–470. Springer US, 2006.
18. B. Kreuter, A. Shelat, B. Mood, and K. R. B. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Proceedings of the 22th USENIX Security Symposium*, pages 321–336, 2013.
19. J. Launchbury, D. Archer, T. DuBuisson, and E. Mertens. Application-scale secure multiparty computation. In *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 8–26. Springer Berlin Heidelberg, 2014.
20. Y. Lindell, K. Nissim, and C. Orlandi. Hiding the input-size in secure two-party computation. In K. Sako and P. Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 421–440. Springer, Heidelberg, Dec. 2013.
21. Y. Lindell and B. Pinkas. A proof of security of Yao’s protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, Apr. 2009.
22. 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*, pages 623–638. IEEE Computer Society Press, May 2014.
23. C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. OblivVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015.
24. P. Mohassel, S. S. Sadeghian, and N. P. Smart. Actively secure private function evaluation. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 486–505. Springer, Heidelberg, Dec. 2014.
25. R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303. ACM Press, May 1997.
26. N. Pippenger and M. J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.
27. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with  $O((\log N)^3)$  worst-case cost. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Heidelberg, Dec. 2011.
28. E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.
29. 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 CCS 13*, pages 299–310. ACM Press, Nov. 2013.

30. The LLVM Project. clang: a C language family frontend for LLVM. <http://clang.llvm.org/>.
31. X. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 15*, pages 850–861. ACM Press, Oct. 2015.
32. X. S. Wang, S. D. Gordon, A. McIntosh, and J. Katz. Secure computation of MIPS machine code. In I. G. Askoxyllakis, S. Ioannidis, S. K. Katsikas, and C. A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 99–117. Springer, Heidelberg, Sept. 2016.
33. X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for secure computation. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 14*, pages 191–202. ACM Press, Nov. 2014.
34. S. Zahur and D. Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. <http://eprint.iacr.org/2015/1153>.
35. Y. Zhang, A. Steele, and M. Blanton. PICCO: a general-purpose compiler for private distributed computation. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 13*, pages 813–826. ACM Press, Nov. 2013.

## A 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 8 for some field  $\mathbb{F}$ . Since most schemes use linear secret sharing where the sharing commutes with linear operations, the addition operation can 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 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.

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

Fig. 8.  $\mathcal{F}_{ABB}$

Keller and Scholl [16] 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 9. 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 10.

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

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

**Fig. 9.**  $\mathcal{F}_{ABBOA}$  extension

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

**Fig. 10.**  $\mathcal{P}_{ABBOA}$

*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.

### A.1 Random Access Machines

Consider 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 3. The first instruction allows to load a constant to a fixed memory address, the second and 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 “PRINT” 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 3.** Instructions of random access machines

## B 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 be implemented using the arithmetic black box with oblivious arrays. Figure 11 shows the desired functionality of the oblivious machine.  $n_c$  and  $n_I$  denote the number of parameters and supported instructions, and  $n_d$  and  $n_p$  denote the size of the data and the code, respectively.

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

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

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

**Private Code:** On input  $(\text{PrivateCode}, i, (c_0, \text{param}_0), \dots, (c_{n_c-1}, \text{param}_{n_c-1}))$  from Party  $P_i$  and  $(\text{privateCode}, i, n_c, n_p)$  from all other parties, check that  $|\text{param}_j| = n_p$  for all  $j$ , and store the code array.  $(c_0, \text{param}_0), \dots, (c_{n_c-1}, \text{param}_{n_c-1})$  denote instructions with parameters.

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

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

**Run:** On input  $(\text{Run}, pc)$  from all parties, execute the following:

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

**Reveal:** On input  $(\text{Reveal}, i, a)$  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  $(\text{WithinBounds})$  to the environment. Otherwise, output  $(\text{OutOfBounds})$  to all parties and the environment and abort.

**Fig. 11.**  $\mathcal{F}_{\text{machine}}$

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 with a certain instruction pattern is defined by the circuits that are computed before, between, and after the memory accesses. The first circuit uses the runtime arguments  $\text{param}$  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  $r_i$  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. Finally, there is also a persistent state between the circuits.



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

**Definition 2 (Instruction).** An instruction with instruction pattern  $\{(n_1, t_1), \dots, (n_L, t_L)\}$  and parameter size  $n_0$  is a list  $\{c_1, \dots, c_{L+1}\}$  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  $\text{param}$ , program counter  $pc$

**Output:** Address of next instruction

state<sub>0</sub>  $\leftarrow$  param

$r_0 \leftarrow \perp$

**for**  $i \in (1, \dots, L)$  **do**

Execute  $c_i$  with input  $(r_{i-1}, pc, \text{state}_{i-1})$  and output  $(a_i, w_i, f_i, \text{state}_i)$ .  $\triangleright |\text{state}_i| = n_i$  and  $|\text{state}_{i+1}| = n_{i+1}$

**if**  $t_i = \text{Read}$  **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<sub>L+1</sub>

$\triangleright |\text{state}_{L+1}| = 1$

---

As an example, we will now explain our implementation in Table 4 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,  $t_1 = \text{Read}$ ,  $t_2 = \text{Read}$ , and  $t_3 = \text{Write}$ .

**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 4 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  $(i, j, k)$  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.

**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.  $\mathcal{F}_{\text{machine}}$  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 scenarios 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. This is clearly inherent to private function evaluation. However, the underlying MPC scheme guarantees that no information leaks during the computation. Therefore, the leakage is limited by the size of the output.

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

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

**Table 4.** Implementation of the random access machine by Cook and Reckhow

## B.1 Implementation Using $\mathcal{F}_{ABBOA}$

We now propose protocol  $\mathcal{P}_{machine}$  in Figure 12 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  $(c_0, \text{param}_0), \dots, (c_{n_c-1}, \text{param}_{n_c-1})$ .

**Private Code:** Initialize an oblivious array  $[C]$  of size  $n_c \cdot (1 + n_p)$  and fill it with  $([c_0], [\text{param}_0]), \dots, ([c_{n_c-1}], [\text{param}_{n_c-1}])$  input to the arithmetic black box  $\mathcal{F}_{ABB}$  by Party  $P_i$ .

**Public Data:** Send  $(\text{WriteArray}([D], \mathcal{F}_{ABBOA}(\text{PublicInput}, a), \mathcal{F}_{ABBOA}(\text{PublicInput}, d)), \mathcal{F}_{ABBOA}(\text{PublicInput}, 1))$  to  $\mathcal{F}_{ABBOA}$ .

**Private Data:** Send  $(\text{WriteArray}([D], [a], [d], \mathcal{F}_{ABBOA}(\text{PublicInput}, 1))$  to  $\mathcal{F}_{ABBOA}$ .

**Run:** The parties execute the following:

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

**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. 12.**  $\mathcal{P}_{machine}$

**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 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 13 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.

**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. 13.**  $\mathcal{S}_{machine}$

**Complexity.** The amount of communication and computation depends on the cost of accessing the oblivious array. Keller and Scholl [16] report an implementation based on Path ORAM [29] 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 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.

We have also implemented Circuit ORAM [31], but we found it to be slower in our context than Path ORAM, despite the improved asymptotic complexity. We assume that this is due to the higher number of rounds of multiplications in Circuit ORAM, which translate to communication rounds in secret sharing-based MPC.

## C 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

```

1  define i64 @match(i64* %YYCURSOR) #0 {
2  entry:
3      %retval = alloca i64, align 8
4      %YYCURSOR.addr = alloca i64*, align 8
5      %YYMARKER = alloca i64*, align 8
6      %yych = alloca i64, align 8
7      store i64* %YYCURSOR, i64** %YYCURSOR.addr, align 8
8      br label %while.body
9
10 while.body:                                ; preds = %entry
11     %0 = load i64** %YYCURSOR.addr, align 8
12     %1 = load i64* %0, align 8
13     store i64 %1, i64* %yych, align 8
14     %2 = load i64* %yych, align 8
15     br label %LeafBlock
16
17 LeafBlock:                                  ; preds = %while.body
18     %SwitchLeaf = icmp eq i64 %2, 97
19     br i1 %SwitchLeaf, label %sw.bb, label %NewDefault
20
21 sw.bb:                                       ; preds = %LeafBlock
22     br label %yy2
23
24 NewDefault:                                  ; preds = %LeafBlock
25     br label %sw.default
26
27 sw.default:                                  ; preds = %NewDefault
28     br label %yy5
29
30 yy2:                                         ; preds = %sw.bb1, %sw.bb
31     %3 = load i64** %YYCURSOR.addr, align 8
32     %incdec.ptr = getelementptr inbounds i64* %3, i32 1
33     store i64* %incdec.ptr, i64** %YYCURSOR.addr, align 8
34     %4 = load i64** %YYCURSOR.addr, align 8
35     %5 = load i64* %4, align 8
36     store i64 %5, i64* %yych, align 8
37     %6 = load i64* %yych, align 8
38     br label %NodeBlock

```

Fig. 15. The regular expression “ab\*[de]” compiled to LLVM code (part one)

```

1 NodeBlock:                                ; preds = %yy2
2   %Pivot = icmp ult i64 %6, 99
3   br i1 %Pivot, label %LeafBlock2, label %LeafBlock4
4
5 LeafBlock4:                                ; preds = %NodeBlock
6   %.off = add i64 %6, -99
7   %SwitchLeaf5 = icmp ule i64 %.off, 1
8   br i1 %SwitchLeaf5, label %sw.bb2, label %NewDefault1
9
10 LeafBlock2:                               ; preds = %NodeBlock
11  %SwitchLeaf3 = icmp eq i64 %6, 98
12  br i1 %SwitchLeaf3, label %sw.bb1, label %NewDefault1
13
14 sw.bb1:                                   ; preds = %LeafBlock2
15  br label %yy2
16
17 sw.bb2:                                   ; preds = %LeafBlock4
18  br label %yy7
19
20 NewDefault1:                              ; preds = %LeafBlock2, ↵
21  %LeafBlock4
22  br label %sw.default3
23
24 sw.default3:                              ; preds = %NewDefault1
25  br label %yy4
26
27 yy4:                                       ; preds = %sw.default3
28  br label %yy5
29
30 yy5:                                       ; preds = %yy4, %sw.↵
31  default
32  %7 = load i64**, %YYCURSOR.addr, align 8
33  %incdec.ptr4 = getelementptr inbounds i64* %7, i32 1
34  store i64* %incdec.ptr4, i64** %YYCURSOR.addr, align 8
35  store i64 0, i64* %retval
36  br label %return
37
38 yy7:                                       ; preds = %sw.bb2
39  %8 = load i64** %YYCURSOR.addr, align 8
40  %incdec.ptr5 = getelementptr inbounds i64* %8, i32 1
41  store i64* %incdec.ptr5, i64** %YYCURSOR.addr, align 8
42  store i64 1, i64* %retval
43  br label %return
44
45 return:                                   ; preds = %yy7, %yy5
46  %9 = load i64* %retval
47  ret i64 %9
48 }

```

Fig. 16. The regular expression “ab\*[de]” compiled to LLVM code (part two)

```

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

```

Fig. 17. The regular expression “ab\*[de]” compiled for the oblivious machine