Secure Computation of MIPS Machine Code

Xiao Wang
University of Maryland
wangxiao@cs.umd.edu

S. Dov Gordon
George Mason University
gordon@gmu.edu

Allen McIntosh
Applied Communication Sciences
amcintosh@appcomsci.com

Jonathan Katz
University of Maryland
jkatz@cs.umd.edu

Abstract

Existing systems for secure computation require programmers to express the program to be securely computed as a *circuit*, or in a *domain-specific language* that can be compiled to a form suitable for applying known protocols. We propose a new system that can securely execute *native MIPS code* with no special annotations. Our system allows programmers to use a language of their choice to express their programs, together with any off-the-shelf compiler to MIPS; it can be used for secure computation of "legacy" MIPS code as well.

Our system uses oblivious RAM for fetching instructions and performing load/store operations in memory, and garbled universal circuits for the execution of a MIPS CPU in each instruction step. We also explore various optimizations based on an offline analysis of the MIPS code to be executed, in order to minimize the overhead of executing each instruction while still maintaining security.

1 Introduction

Systems for secure two-party computation allow two parties, each with their own private input, to evaluate an agreed-upon program on their inputs while revealing nothing other than the result of the computation. This notion originated in the early 1980s [Yao86], and until recently was primarily of theoretical interest, with research focusing entirely on the design and analysis of low-level cryptographic protocols. The situation changed in 2004 with the introduction of Fairplay [MNPS04], which provided the first implementation of a protocol for secure two-party computation in the semi-honest setting. Since then, there has been a flurry of activity implementing two-party protocols with improved security and/or efficiency [EFLL12, PSSW09, HKS+10, HEKM11, KSS12, HFKV12, LHS+14, KMSB13, RHH14, LWN+15, ZE15, LR15, AMPR14].

Many (though not all) of these implementations provide an *end-to-end system* that, in principle, allows non-cryptographers to write programs that can automatically be compiled to some intermediate representation (e.g., a boolean circuit) suitable for use by back-end protocols that can securely execute programs expressed in that representation. In practice, however, such systems have several drawbacks. First of all, the user cannot write the program in a language of their choice; just as there is no programming language that is best for every application, there is no domain-specific language for secure computation that is best for every purpose. Second, existing domain-specific languages [HFKV12, LHS+14, LWN+15, RHH14, ZE15] can be hard to learn and use, or are simply

limited in terms of expressiveness. For example, Secure Computation API (SCAPI) [EFLL12] does not provide a high-level language to specify the function to compute. Sharemind [BLW08] fails to support conditional statements branching on private variables according to the user manual. Obliv-C [ZE15], an extension of C, requires the programmer to use special annotations along with an extended set of keywords for secret values and branching statements. Even if the language is a subset of a standard language (such as ANSI C [HFKV12]), the programmer must still be aware of limitations and which features of the language to avoid; moreover, legacy code will not be supported. Finally, compilers from high-level languages to boolean circuit representations (or other suitable representations) can be slow; as discussed in detail in Section 5.2, some previous compilers [KMSB13, KSS12] require more than 2000 seconds just to compile a program for matrix multiplication of dimension 16 into a circuit representation, not even including any cryptographic operations. Furthermore, most compilers requires recompiling the program when the input size changes.

Motivated by these drawbacks, we explore in this work the design of a system for secure execution, in the semi-honest setting, of *native MIPS machine code*. That is, the input program to our system—i.e., the program to be securely computed—is expressed in MIPS machine code, and our system securely executes this code. We accomplish this via an emulator that securely executes each instruction of a MIPS program; the emulator uses oblivious RAM [GO96] for fetching the next instruction as well as for performing load/store operations in memory, and relies on a universal garbled circuit for executing each instruction. Our system addresses all the problems mentioned earlier:

- Programmers can write their programs in a language of their choice, and compile them using any compiler of their choice, so long as they end up with MIPS machine code. Legacy MIPS code is also supported.

- Because our system does not require compilation from a domain-specific language to some suitable intermediate representation, such as boolean circuits, we can use fast, off-the-shelf compilers and enjoy the benefits of all the optimizations such compilers already have. The number of instructions we securely execute is identical to the number of instructions executed when running the native MIPS code (though, of course, our emulator introduces overhead for each instruction executed).

Our code is open sourced online [1].

Our primary goal was simply to develop a system supporting secure execution of native MIPS code. In fact, though, because of the way our system works—namely, via secure emulation of each instruction—we also gain the benefits of working in the RAM model of computation (rather than in a boolean-circuit model), as in some previous work [GKK+12, LHS+14, LWN+15]. In particular, the total work required for secure computation of a program using our system is proportional (modulo polylogarithmic factors) to the actual number of instructions executed in an insecure run of that same program on the same inputs; this is not true when using a circuit-based representation, for which all possible program paths must be computed even if they are eventually discarded. Working in the RAM model even allows for computation time sublinear in the input length [GKK+12] (in an amortized sense); we show an example of this in Section 5.4.

[1] https://github.com/wangxiao1254/Secure-Computation-of-MIPS-Machine-Code

Performance. Our goal is generality, usability, and portability; we expressly were **not** aiming to develop a yet-more-efficient implementation of secure computation. Nevertheless, for our approach to be viable it must be competitive with existing systems. We describe a number of optimizations that rely on an offline analysis of the (insecure) program to be executed; these can be done before the parties' inputs are known and before execution of any protocol between them begins. We view these optimizations as one of the main contributions of our work, as they improve the performance by as much as a factor of 30× on some programs, bringing our system to the point where it is feasible. In fact, for certain applications we are only ∼25% slower than ObliVM [LWN+15] (see Section 5.4).

Trade-off between efficiency and usability. Our work explores part of the spectrum between *efficiency* and *usability* for secure-computation systems. Most work in this area has concentrated on the former, focusing on optimizing the back-end protocol, implementation aspects, or improved compilation time. Here, we are expressly interested in maximizing usability, envisioning, e.g., a non-expert user maintaining a large code base, perhaps written in several languages, who occasionally wishes to run some of this code securely. Such a user might gladly sacrifice run-time efficiency in order to avoid re-writing their code in the domain-specific language of the moment. To support this level of generality, the only feasible approach is to securely compute on a low-level language. One goal of our work is to explore how much efficiency must be sacrificed in order to achieve this level of generality.

Related and concurrent works. Songhori et al. [SHS+15] explored using hardware-optimization techniques to reduce the size of boolean circuits and demonstrated a circuit containing MIPS-I instructions. However, the goal of their work was different: they aim to minimize the size of a single universal circuit for private function evaluation (PFE), while we aim to optimize the emulation of an entire public MIPS program. In particular, they do not investigate optimizations to accelerate execution of the program, something that is a key contribution of our work. Fletcher et al. [FDD12] designed an interpreter based on the Turing machines, using static analysis on the program to improve the efficiency. Their levelization technique shares some similarities to some of our techniques. However their system does not support general RAM computation, therefore is simpler than our setting.

Concurrently, Keller [Kel15] recently built a system that executes C code over the SPDZ protocol. Their high level idea is similar to our *basic system*: using a universal circuit for ALU and ORAM for memory access. However, they didn't explore how to use static analysis to further accelerate the system, as in our *optimized system* described in Section 4.

2 Preliminaries

We briefly describe some background relevant to our work.

Secure computation and garbled circuits. Protocols for two-party computation allow two parties, each with their own private input, to compute some agreed-upon function on their inputs while revealing nothing to either party other than the result of the computation. In this paper we exclusively focus on the semi-honest model, where both parties are assumed to execute the protocol correctly, but each may try to infer additional information about the other party's input based on their own view of the protocol transcript.

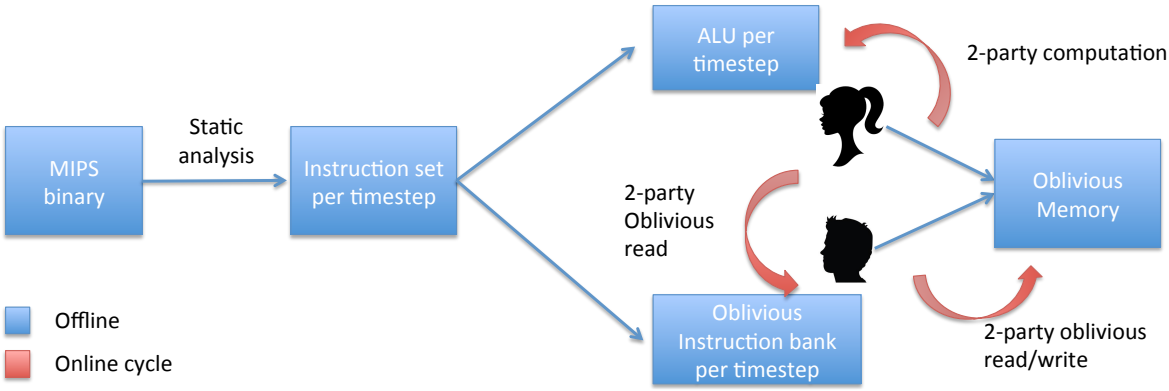


Figure 1: Our system takes native MIPS code as input, and performs offline analysis resulting in a garbled circuit (corresponding to execution of the MIPS CPU) for each time step. During online execution, the parties execute these garbled circuits on inputs fetched using oblivious RAM.

Our emulator uses as a building block Yao's garbled-circuit protocol [Yao86] for secure two-party computation, which assumes the function to be computed is represented as a boolean circuit. At a high level, the protocol works as follows: one party, acting as a *garbled-circuit generator*, associates two random cryptographic keys with each wire in the circuit. One of these keys will represent the value 0 on that wire, and the other will represent the value 1. The circuit generator also computes a garbled table for each gate of the circuit; the garbled table for a gate g allows a party who knows the keys associated with bits b_L, b_R on the left and right input wires, respectively, to compute the key associated with the bit $g(b_L, b_R)$ on the output wire. The collection of garbled gates constitutes the garbled circuit for the original function. The circuit generator sends the garbled circuit to the other party (the *circuit evaluator*) along with one key for each input wire corresponding to its own input. The circuit evaluator obtains one key for each input wire corresponding to *its* input using oblivious transfer. Given one key per input wire, the circuit evaluator can "evaluate" the garbled circuit and compute a single key per output wire.

If the circuit generator reveals the mapping from the keys on each output wire to the bits they represent, then the circuit evaluator can learn the actual output of the original function. However, it is also possible for the circuit generator to use the output wires from one garbled circuit as input wires to a subsequent garbled circuit (for some subsequent computation being done), in which case there is no need for this decoding to take place. In fact, this results in a form of "secret sharing" that we use in our protocol; namely, the sharing of a bit b will consist of values (w^0, w^1) held by one party, while the other party holds w^b.

Oblivious RAM (ORAM). Oblivious RAM [GO96] allows a "client" to read/write an array of data stored on a "server," while hiding from the server both the the data and the data-access pattern. By default, ORAM only hides information from the server; to hide information from both parties, we adopt the method of Gordon et al. [GKK+12] and share the client state between the two parties, who can then use the shares as input to a secure computation that outputs a memory address to read, while also updating both users' shares of the state.

In this work, we use ORAM to store both the MIPS program (so parties are oblivious of the instruction being fetched) as well as the contents of main memory (so parties are oblivious about the location of memory being accessed). Note that in the former case, the data is public and read-

only, whereas in the latter case the data needs to be kept hidden from both parties (in general), and may be both read and written. We refer to the ORAM storing instructions as the "instruction bank," and the ORAM storing data as the "memory bank."

Security of our emulator. Although we do not provide any proofs of security in this work, an argument as in [GKK+12] can be used to prove that our construction is secure (in the semi-honest model) with respect to standard definitions of security [Gol04], with one important caveat: our system leaks the total number of instructions executed in a given run of the program. This is a consequence of the fact that we allow loop conditions based on private data, something that is simply disallowed in prior work. Leaking the running time may leak information about the parties' private inputs; note, however, that this can easily be prevented by using padding to ensure that the running time is always the same. In section 4.3, we will discuss how to reduce this leakage while improving the efficiency at the same time.

3 Basic System Design

In this section we describe the basic design of our system. We describe the overall workflow in Section 3.1. In Section 3.2, we review relevant aspects of the MIPS architecture, and in Section 3.3, we give a high-level overview of how our system works. We provide some low-level details in Sections 3.4 and 3.5. We defer until Section 4 a discussion of several important optimizations that we apply.

3.1 Overall Workflow

Our system enables two parties to securely execute a program described in a MIPS code. Our system works in the following steps:

- It first performs an offline static analysis of the MIPS code to produce a set of CPU circuits and instruction banks, one for each step of the computation. In our basic system described in this section, the offline analysis is rather simple; a more complex analysis, described in Section 4, can be used to improve performance.
- During the online phase, the two parties securely execute each instruction, using ORAM to access the appropriate inputs at each step.

As our back-end for secure computation (namely, generation/execution of the garbled circuits, and fetching of inputs using ORAM) we use ObliVM [LWN+15]. However, other frameworks [DSZ15, SHS+15, ZE15] could also potentially be used (in conjunction with other ORAM implementations).

3.2 MIPS Architecture

A MIPS program is an array of instructions, each 32 bits long. In the basic MIPS instruction set (i.e., in MIPS I), there are about 60 instruction types, including arithmetic instructions, memory-related instructions, and branching instructions. Instruction types refer to the operations performed during a CPU cycle. On the other hand, an instruction consists of instruction type and operands. For example, `ADD` is an instruction type, but `ADD $1, $2, $2` is an instruction.

Types	Instruction Type									
R Type	ADDU MTLO	MOVZ MFHI	MOVN MTHI	SLLV MULT	SRLV SUBU	MFLO SLTU	SLL OR	SRL XOR	SRA NOR	AND DIV
I Type	BGEZAL ADDIU	SLTIU BLTZAL	XORI BGEZ	ANDI BLTZ	BLEZ BGTZ	JR BEQ	ORI LUI	BNE		
J Type	J	JAL								

Table 1: **Set of instruction types currently supported in our system. More instructions can be added easily.**

For our purposes, we can view the state of the MIPS architecture during program execution as consisting of (1) the program itself (i.e. the array of instructions), (2) the values stored in a set of 32-bit registers, which include 32 general-purpose registers plus various special registers including one called the *program counter*, and (3) values stored in main memory. To execute a given program, the input(s) are loaded into appropriate positions in memory, and then the following steps are repeated until a special exit instruction is executed:

- Instruction fetch (`IF`): fetch the instruction according to the program counter.
- Instruction decode (`ID`): fetch 2 registers according to the instruction.
- Execute (`EX`): execute the instruction and update the program counter.
- Memory access (`MEM`): perform load/store operations on memory, if required (depending on the instruction).
- Write back (`WB`): write a value to one register.

3.3 Overview of Our System

At a high level, two parties use our system to securely execute a MIPS program by maintaining *secret shares* of the current state, and then *updating* their shares by securely emulating each of the steps listed above until the program terminates. We describe each of these next.

We currently support about 37 instruction types (see Table 1), which are sufficient for all the programs used in our experiments. It is easy to add instruction types to our system as needed, using 2–3 lines of code (in the ObliVM framework) per instruction. In our basic system described here, every supported instruction is included in the garbled circuit for every step, thus increasing the run-time of each emulated MIPS cycle. In our optimized system described in Section 4, only instructions that can possibly be executed at some step are included in the garbled circuit for that step, so there is no harm in including support for as many instructions types as desired.

Secret sharing the MIPS state. As mentioned previously, the MIPS state contains the array of program instructions, registers, and memory; all three components are secret shared between the two parties and, in addition, the program instructions and memory are stored in (separate) ORAMs. (Even though the program instructions are known to both parties, it is important that neither party learns which instruction is fetched in any instruction cycle, as this leaks information about the inputs.) The registers could, in principle, also be stored in ORAM, but since there are only 32 registers a trivial ORAM (i.e., a linear scan over all registers) is always better.

By default, all components are secret shared using the mechanism described in Section 2. Although this results in shares that are 80–160× larger than the original value (because ObliVM creates garbled circuits with 80-bit keys), this sharing is more efficient for evaluating garbled circuits on those shared values. However, when the allocated memory is larger than 12MB, we switch to a more standard XOR-based secret-sharing scheme, adding an oblivious-transfer step and an XOR operation inside the garbled circuit to reconstruct the secret.

Secure emulation. The parties repeatedly update their state by performing a sequence of secure computations in each MIPS instruction cycle. For efficiency, we reordered the steps described in the previous section slightly. In the secure emulation of a single cycle, the parties:

1. Obliviously fetch the instruction specified by the shared program counter (the `IF` step).
2. Check whether the program has terminated and, if so, terminate the protocol and reconstruct the output.
3. Securely execute the fetched instruction, and update the registers appropriately (this corresponds to the `ID`, `EX`, and `WB` steps).
4. Obliviously access/update the memory and securely update a register if the instruction is a load or store operation (this corresponds to the `MEM` and `WB` steps).

We stress that the parties should not learn whether they are evaluating an arithmetic instruction or a memory instruction, and must therefore execute steps 3 and 4 in every cycle, even if the step has no effect on the shared MIPS state. Our improved design in Section 4 provides a way of securely bypassing step 4 on many cycles.

3.4 Setup

Before executing the main loop, we load the MIPS code into the (shared) instruction memory and the users' inputs into the (shared) main memory.

Loading the MIPS code. In our baseline system design, we load the full program (i.e., array of instructions) into an ORAM. Therefore, when emulating each step we incur the cost of accessing an ORAM containing instructions from the entire program. In Section 4, we describe improvements to this approach.

In our current implementation, we do not load any code executed before `main()` is called, e.g., library start-up code, code for managing dynamic libraries, or class-constructor code. The latter is not needed for executing MIPS programs compiled from C code, but would be needed for executing MIPS code generated from object-oriented languages. Note, however, that such operations are data-independent, and can be simulated by the parties locally. Adding support for loading such code would thus be easy to incorporate in our system.

Loading user inputs. We assume a public upper bound on the input size of each party. Each party starts with their input in a local file. When emulation begins, the parties initialize an empty ORAM supporting the maximum input sizes, and the parties' inputs are secret shared and written to some agreed-upon (non-overlapping) segments of memory. The parties also initialize their shares of the register space with pertinent information such as the address and length of the input data. Since no annotation is used, we need to find a way to specify which party each input belongs to.

Secure computation of MIPS code

Input: reg[], pc, $\mathsf{ORAM}_{\mathsf{inst}}$[], $\mathsf{ORAM}_{\mathsf{MEM}}$[]

Computation:

1. inst := FetchInstruction(pc, $\mathsf{ORAM}_{\mathsf{inst}}$)
2. terminateBit := testTerminate(inst, reg)
3. if (terminateBit) GOTO line 7
4. ALU(inst, pc, reg)
5. MEM(inst, reg, $\mathsf{ORAM}_{\mathsf{MEM}}$)
6. GOTO line 1
7. Reconstruct the output (in reg or $\mathsf{ORAM}_{\mathsf{MEM}}$).

Figure 2: Overview of secure computation of a MIPS program. Boxed lines are executed outside of secure computation.

In our system, each party organizes their input as an array, which is passed to the function to compute in an fixed order: generator's array comes first; evaluator's array comes second, followed by the length of two arrays.

3.5 Main Execution Loop

We use $\mathsf{ORAM}_{\mathsf{inst}}$[], $\mathsf{ORAM}_{\mathsf{MEM}}$[], reg[], pc, and inst to denote, respectively, the (shared) instruction bank, (shared) memory bank, (shared) registers, (shared) value of the program counter, and (shared) current instruction. As shown in Figure 2, secure execution of a MIPS program involves repeated calls of three procedures: instruction fetch, ALU computation, and memory access.

Instruction fetch. In the basic system, we put the entire MIPS program into an ORAM. Therefore, fetching the next instruction is simply an ORAM lookup.

ALU computation. The MIPS ALU is securely computed using a universal garbled circuit. As shown in Figure 3, this involves five stages:

1. Parse the instruction and get fields including operation code, function code, register addresses, and the immediate value. (We use inst[s:e] to denote the s-th bit to the e-th bits of inst.)
2. Retrieve values reg_rs = reg[rs] and reg_rt = reg[rt] from the registers.
3. Perform arithmetic computations.
4. Update the program counter.
5. Write the updated value back to the registers.

The first step is free due to the secret sharing we use and the fact that here we are using a circuit model of computation. The fourth step is very cheap. The second and fifth steps require 3 accesses to the register space in total, which we have implemented using a circuit with 3552 AND gates.

Secure computation of the MIPS ALU

Input: inst, pc, reg[]

ALU:

1. rs := inst[21:25], funct := inst[0:5],rd := inst[11:15]
   rt := inst[16:20], op := inst[26:31],imm := inst[0:15], ...
2. reg_rs := reg[rs], reg_rt := reg[rt]
3. if op == R_Type and funct == ADD
   reg_rd := reg_rs + reg_rt
   else if op == I_Type and funct == ADDI
   reg_rt := reg_rs + immediate
   else if ... // more arithmetic operations
4. if op == R_Type then pc := pc + 4
   else if op == I_Type and funct == BNE
   if reg_rs! = reg_rt
   pc := pc + 4 + 4 × imm
   else if ... // more cases that update pc
5. if op == R_Type then reg[rd] := reg_rd
   else reg[rt] := reg_rt

Figure 3: This functionality takes the current instruction, program counter, and registers as input. Depending on the type of the instruction, it performs computation and updates the registers, as well as updating the program counter.

Memory access. Memory-related instructions can either load a value from memory to a register, or store a value from a register to memory. As shown in Figure 4a, in order to hide the instruction type, every memory access requires one read and one write to the memory ORAM, as well as a read and a write to the registers. The cost of this component depends on how large the memory is, and is often the most expensive part of the entire computation.

Checking for termination. In our basic implementation, we execute a secure computation on each cycle in order to determine whether the program has terminated. (See Figure 4b.) This is done by checking the current instruction and the contents of the final register (used for storing the return value). Revealing this bit to both parties requires one round trip in each instruction cycle.

4 Improving the Basic Design

The construction described in Section 3 requires us to perform a secure emulation of the full MIPS architecture for every instruction cycle. Even if we restrict our system to only include a small number of instruction types, we still have to execute many unnecessary instructions in every step: if a single expensive instruction appears *anywhere* in the program, our basic system would execute this expensive instruction at every step, even though the result is usually ignored. Even worse is the fact that the presence of load/store instructions necessitates expensive accesses to the memory ORAM in every instruction cycle.

Secure memory access

Input: inst, reg[], $\mathsf{ORAM}_{\mathsf{MEM}}$[]
MEM:

1. rs:=inst[21:25], op:=inst[26:31], rt:=inst[16:20], imm:=inst[0:15]
2. addr := imm+reg[rs]
3. If op is a load operation code: reg[rt] := $\mathsf{ORAM}_{\mathsf{MEM}}$[addr]
4. If op is a store operation code: $\mathsf{ORAM}_{\mathsf{MEM}}$[addr] := reg[rt]

(a) This functionality takes the instruction, the registers, and the memory as input and accesses the memory and registers according to the instruction.

Testing termination

Input: inst, reg[31]
testTerminate:

1. terminate := false
2. If inst is `BEQ $0,$0,-1` terminate := true
3. If inst is `jr $31` and reg[31] == 0 terminate := true
4. Reveal the bit terminate.

(b) This functionality takes the current instruction and the registers as input. It returns true if these indicate program termination.

4.1 Mapping Instructions to Steps

We improve the efficiency of our system by identifying unnecessary computation. First, we perform static analysis of the MIPS binary code and compute, for every step of the program execution, the set of instructions that might possibly be executed in that step. Then, using this information, we automatically generate a small instruction bank and ALU circuit tailored for each time step. This allows us to improve performance, without affecting security in various ways.

Computing instruction sets for each time step. To compute a set of instructions that might be executed at each time step, we walk through the binary code, spawning a new thread each time we encounter a branching instruction. Each thread steps forward, tagging the instructions it encounters with an increasing time step, and spawning again when it encounters a branch. We terminate our analysis if all threads halt, or if the set of all instructions tagged with current time step L is the same as the set of instructions tagged with some previous time step $k < L$. It is easy to verify that one of these two conditions will eventually be met. Now, the set of instructions that should be executed at time step $i < L$ contains all instructions tagged with time step i.

During the execution, our emulator chooses a circuit according to the following deterministic sequence of time steps: $1, 2, ..., L, k+1, ..., L, k+1, ..., L, ...$, until the termination condition is satisfied. In Section 5.2, we will discusse in detail how long such static analysis takes on programs of different sizes.

To illustrate this procedure, we provide a very simple example in Figure 5a. Although there are eight instructions in the code snippet, at most two instructions can possibly be executed in any time step. When the code contains loops, as shown in Figure 5b, a single instruction might appear in multiple time steps. In this case, our analysis will only terminate when we repeat some prior state, resulting in an instruction set that is identical to one that we previously constructed. In particular, the set of instructions for time step $2k+4$ is the same as the one for time step $k+3$.

This analysis does not result in an optimal assignment of instructions to time steps, because it ignores data values. We leave it to future work to explore better methods of performing the mapping of instructions to time steps. On the other hand, because of this it is easy to see that no private information is leaked since the set of instructions corresponding to some time step t includes all possible instructions that could ever be executed at step t for any possible set of inputs.

Example 1

```
main:
 MULT $1, $2, $3   //step 1
 BNE $1, $2, else  //step 2
 Instruction 1     //step 3
 Instruction 2     //step 4
 J endif           //step 5
else:
 Instruction 3     //step 3
 Instruction 4     //step 4
 Instruction 5     //step 5
endif:
```

(a) Assigning instructions to time steps. `MULT` instruction is not included in any ALU circuit but step 1.

Example 2

```
main:
 ADD $1,$2,$3 //step 1
lo:
 Ins. 1       //step 2, k+3, 2k+4
 ...
 Ins. k       //step k+1, 2k+2
 BNE $1,$2,lo //step k+2, 2k+3
post-lo:
 Ins. k+1     //step k+3, 2k+4
 ...
 Ins. 2k+1    //step 2k+3
```

(b) An example demonstrating how we map instructions to time steps in a program with loops.

Instruction mapping makes sure that for each step only the instruction types that can possibly be executed in that step will be included. Therefore, in the optimized system, adding support for more instructions *will not impact the performance*. In particular, although we have not implemented the full MIPS instruction set, doing so would have no impact on the performance results described in Section 5 because the unnecessary instructions are automatically excluded by our emulator.

Constructing smaller instruction banks. After performing the above analysis, we can initialize a set of instruction banks in the setup stage, one for each time step, to replace the single, large instruction bank used in Section 3. When fetching an instruction during execution, we can simply perform an oblivious fetch on the (smaller) instruction bank associated with the current time step.

When we employ this optimization, using naïve ORAM to store the set of possible instructions for each time step becomes inefficient. Originally, instructions were in contiguous portions of memory, so N instructions could be placed into an ORAM of size N. Now, each instruction set contains only a small number of instructions, say $n < N$, while their address values still span the original range. If we use ORAM to store them, its size would have to be N instead of n. Therefore, we use an oblivious key-value store for the set of instructions at each time step. Since the size of each instruction bank is very small for the programs we tested, we implemented an oblivious key-value structure using a simple linear scan; for larger programs with more instructions it would be possible to design a more-complex oblivious data structure with sub-linear access time.

Constructing smaller ALU circuits. Once we have determined the set of possible instructions for a given time step, we can reduce the set of *instruction types* required by that time step. In the offline phase, before user inputs are specified, we generate a distinct garbled ALU circuit for each time step (using ObliVM) supporting exactly the set of possible instructions in that time step. During online execution, our emulator uses the appropriate garbled ALU circuit at each time step.

Skipping unnecessary memory operations. The same idea also allows us to reduce the number of memory operations. There are two types of memory operations: (1) store operations that read a value from a register and write it to memory, and (2) load operations that read a value from memory and write it to a register. When performing the static analysis, we compute two flags for each time step, indicating if any load or store operation could possibly occur in that step. During

the run-time execution, our emulator skips the load/store computation depending on the values of these flags.

Improving accesses to the register space. We can additionally improve the efficiency of register accesses. Since register values are hard-coded into the instructions, they can be determined offline, before the user inputs are specified. (This is in contrast to memory accesses, where the addresses are loaded into the registers and therefore cannot be determined at compile time.) For example, the instruction `ADD $1, $2, $4` needs to access registers at location 1, 2, and 4. During the offline phase, we compute for each time step the set of all possible register accesses at that time step. Then, in the online phase, only those registers need be included in the secure emulation for that time step.

4.2 Padding Branches

The offline analysis we just described provides substantial improvements on real programs. For example, as we show in Section 5.3 for the case of set intersection, this analysis reduces the cost of instruction fetch by 6× and reduces the ALU circuit size by 1.5×. This is because the full program has about 150 instructions, while the largest instruction bank after performing our binary analysis contains just 31 instructions; for more than half of the time steps, there are fewer than 20 possible instructions per time step. On the other hand, there is only a small reduction in the time spent performing load and store operations, because these operations are possible in nearly every time step. Load and store operations are by far the most costly instructions to emulate. For example, reading a 32-bit integer from an array of 1024 32-bit integer requires $43K$ AND gates. So it is important to reduce the number of times we unnecessarily perform these operations.

Before presenting further improvements, it is helpful to analyze the effect of what has already been described. Consider again the simple example in Figure 5b. Here, with only a single loop, it is easy to calculate how many instructions will be assigned to any particular time step. If the loop has k instructions, and there are n instructions following the loop, then in some time step we might have to execute any of $n/k + 1$ instructions. This should be compared with the worst-case, where we perform no analysis and simply assume that we could be anywhere among the $n+k$ instructions in the program. When k is large and n is small—i.e., if most of the computation occurs inside this loop—our binary analysis provides substantial savings.

Unfortunately, this example is overly simplistic, and in more realistic examples our binary analysis might not be as effective. Let's consider what happens when there is a branch inside the loop, resulting in two parallel blocks of lengths k_1 and k_2. If the first instruction in the loop is reached for the first time at time step x, then it might also be executed at time steps $x+k_1, x+k_2, x+k_1+k_2, \ldots$, and, more generally, at every time step $x+(i\cdot k_1)+(j\cdot k_2)$ for $i, j \in \mathbb{Z}$. If k_1 and k_2 are relatively prime, then every $i \cdot k_1$ time steps, and every $j \cdot k_2$ time steps, we add another instruction to the instruction set. It follows that in fewer than $k_1 \cdot k_2$ time steps overall, we can be anywhere in the loop! Furthermore, at that point we might exit the loop on any step, so that after executing fewer than $k_1 \cdot k_2 + n$ time steps, we might be anywhere from the start of the loop until the end of the program, and we no longer benefit from our analysis at all.

This motivates the idea of padding parallel blocks with NOPs so the length of one is a multiple of the other. Using the same example of a single if/else statement inside a loop, if the two branches have equal length (i.e., $k_1 = k_2$), then at any time step we will never have more than two instructions from inside the loop—one from each branch—assigned to the same time step. We provide further

discussion about the performance improvement in Section 5, using set intersection as an example.

4.3 Checking Termination Less Frequently

In our basic system, we test for termination of the program in every instruction cycle, which incurs a round of communication each time. This overhead becomes significant, especially after we performed the optimizations mentioned in the previous sections. For example, for a program with T cycles, our basic system needs $(r_o+1)T$ roundtrips, where r_o is the number of roundtrips needed by an ORAM access.

In order to avoid such overhead, we modified the system to check for termination only every C instruction cycles, for C a user-specified constant. In every cycle, the parties still compute shares of the bit indicating if the program has terminated, but they do not reconstruct that bit in clear. Instead, memory and register accesses take this bit as input, and do not update the memory or registers if the program has terminated. This ensures that the MIPS state, including registers and memory, will not change after termination, even if additional cycles are executed.

Note that the parties execute up to $C-1$ extra instruction cycles, but the amortized cost of checking for termination is decreased by a factor of C. One can thus set C depending on the relative speeds of computation and communication to minimize the overall running time. Now for a program with T cycles, and n_o memory accesses, the total number of roundtrips is $n_o \times r_o + \lceil T/C \rceil$, where $n_o < T$. In addition to reducing number of roundtrips, such optimization also reduces the leakage. Instead of leaking total number of cycles T, only $\lceil T/C \rceil$ is leaked.

5 Performance Analysis

5.1 Experimental Setup

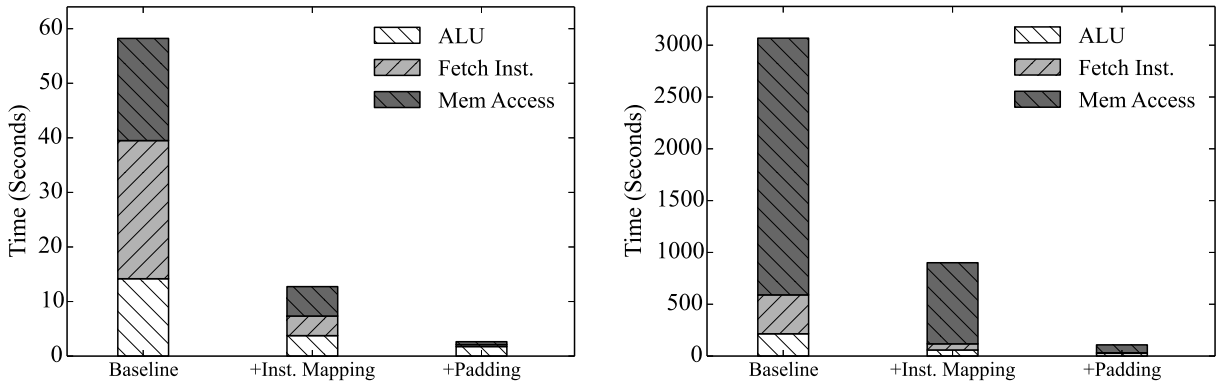
Our emulator takes MIPS binaries as input and allows two parties to execute that code securely. All binaries we used were generated from C code using an off-the-shelf GCC compiler. When specified, we added NOPs to the binary by hand to explore the potential speedup introduced by padding; for all other examples, including Hamming distance, binary search, decision trees, and Dijkstra's shortest-path algorithm, we directly used the binaries generated by the GCC compiler without adding any padding.

All times reported are based on two machines of type `c4.4xlarge` with 3.3GHz cores and 30GB memory running in the same region of an Amazon EC2 cluster. We did not use parallel cores, except possibly by the JVM for garbage collection. In all our results, we report the time used by the circuit generator, which is the bottleneck in the garbled-circuit protocol.

Metrics. Since our emulator uses ALU circuits and instruction banks with different sizes at each time step, and since it may skip memory operations altogether in some time-steps, the cost varies between different time-steps. Therefore, we report the total cost of execution, amortized over all time-steps. To be more specific, we report

- The average number of AND gates per cycle, i.e.,

$$\frac{\text{Total number of AND gates}}{\text{number of cycles}}.$$



(a) Each party's input has 64 32-bit integers.

(b) Each party's input has 1024 32-bit integers.

Figure 6: Run-time decomposition for computing set-intersection size.

- The average number of seconds per cycle, i.e.,

$$\frac{\text{Total number of seconds}}{\text{number of cycles}}.$$

- The total number of times ORAM access is performed.

5.2 Time for Static Analysis and Compilation

Our system takes MIPS binary code, generated by an off-the-shelf compiler, as input. In all the experiments we ran, our system used less than 0.6 seconds for the static analysis mapping instructions to time steps, including generation of the code for the CPU circuits for all time steps. Following this step, we run the ObliVM compiler on the code for these CPU circuits; this never took more than 1.5 seconds for all circuits in any of our examples. Note that compilation time does not include any cryptographic operations; all cryptographic operations, e.g., garbled circuits, oblivious transfer, etc, are all done at runtime, reported separately in the following subsections.

If the program is written in a way such that input size is not fixed at compile time, then we can perform our static analysis and optimization on the binary without prior knowledge of input size. This means that: (1) We only need to perform our static analysis once for all input sizes. (2) The running time of our static analysis is independent of the input size.

The compilation time in our system is within the same range or smaller than systems that report compilation time. For example, two compilers by Kreuter et al. [KMSB13, KSS12] both take more than 2000 seconds to compile a circuit computing matrix multiplication of dimension 16. The time is even higher for larger matrices. In our case, the compilation time is within 2.0 seconds, independent of the size of the matrices.

5.3 The Effect of Our Optimizations

In this section, we explore the impact of different optimizations on the performance of our emulator. We consider the cost for each component of our emulation, i.e., Instruction Fetch, ALU Computation, and Memory Access. We compare three different approaches:

Input size		Memory Access	Instruction Fetch	ALU Computation	Average per Cycle
64	Baseline	9216 (13.57ms)	11592 (17.37ms)	6727 (10.14ms)	27535 (40.97ms)
	+Inst. Mapping	2644 (3.85ms)	1825 (2.57ms)	1848 (2.68ms)	6317 (9.61ms)
	+Padding	258 (0.38ms)	173 (0.25ms)	838 (1.25ms)	1269 (2.37ms)
256	Baseline	21933 (32.57ms)	11592 (17.33ms)	6727 (10.08ms)	40252 (59.8ms)
	+Inst. Mapping	6474 (9.61ms)	1920 (2.67ms)	1885 (2.69ms)	10279 (15.48ms)
	+Padding	622 (0.91ms)	173 (0.25ms)	840 (1.21ms)	1635 (2.87ms)
1024	Baseline	76845 (114.69ms)	11592 (17.37ms)	6727 (9.96ms)	95164 (142.02ms)
	+Inst. Mapping	24479 (36.23ms)	1944 (2.75ms)	1895 (2.72ms)	28318 (42.21ms)
	+Padding	2335 (3.53ms)	173 (0.25ms)	841 (1.22ms)	3349 (5.51ms)

Table 2: Number of AND gates and running time, per cycle, for computing set-intersection size. Sets of 32-bit integers with different sizes are used. Input size shows number of elements from each party.

Input size per party		Memory Access	Instruction Fetch	ALU Computation	Setup	Total Cost
64 Elements	Baseline	18.72 s	25.32 s	14.18 s	0.13 s	58.35 s
	+Inst. Mapping	5.39 s	3.59 s	3.75 s	0.13 s	12.86 s
	+Padding	0.54 s	0.35 s	1.75 s	0.13 s	2.77 s
256 Elements	Baseline	175.88 s	93.59 s	54.45 s	0.17 s	324.09 s
	+Inst. Mapping	51.87 s	14.4 s	14.54 s	0.17 s	80.98 s
	+Padding	4.91 s	1.33 s	6.55 s	0.17 s	12.96 s
1024 Elements	Baseline	2477.39 s	375.24 s	215.19 s	0.37 s	3068.19 s
	+Inst. Mapping	782.5 s	59.43 s	58.65 s	0.37 s	900.94 s
	+Padding	76.31 s	5.46 s	26.31 s	0.37 s	108.45 s

Table 3: **Total running time for computing the size of a set intersection.**

- **Baseline.** This is the basic system (cf. Section 3) with no static analysis.
- **+Instruction Mapping.** For each time-step we compute the set of possible instructions at that time-step, and use a smaller instruction bank and ALU circuit for each cycle, reduce register accesses, and bypass loads/stores whenever possible, as described in Section 4.1.
- **+Padding.** The program is padded (manually) with NOPs as described in Section 4.2.

For evaluation we use a program that computes the size of the intersection of two sets. Each party has an array of 32-bit integers as input and wants to compute how many elements are shared by both parties. The C code used to generate the MIPS binary is included in Figure 9. In Figures 6a and 6b we show the decomposition of the running time when the input from each party is 64 and 1024 integers, respectively. We do not include the setup time in the figure because it is too small relative to the overall running time. However, we report the exact running time, including the setup time, in Table 3. For all running times shown, including the baseline, we incorporated the optimization that checks termination less frequently, as described in Section 4.3.

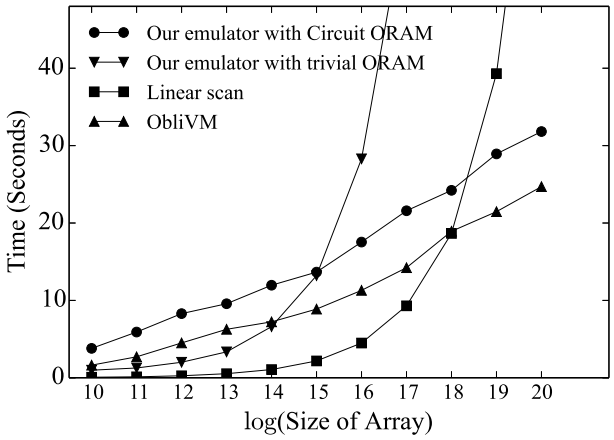


Figure 7: Comparing the performance of secure binary search. One party holds an array of 32-bit integers, while the other holds a value to search for.

log(size of the array)	10	12	14	16	18	20
Baseline System	150	180	210	230	260	290
+Inst. Mapping	11	13	15	17	19	21

Table 4: Number of memory accesses for binary search with different length of arrays.

As shown in Table 3, using static analysis decreases the time used for memory access by 3×, the cost of instruction fetching by 6×, and the time for ALU computation by 3.5×. Using padding gives a further 10× improvement on both instruction-fetch and memory-access time, along with an additional 2× speedup in ALU computation. In total, our optimizations achieve a roughly 28× speedup compared to our baseline system. The cost of instruction fetching is reduced by 67×, the cost of memory access is reduced by 33×, and the cost of ALU computation is cut by 8×.

If we compare Figure 6a, where the input size is 64, and Figure 6b, where the input size is 1024, we find that the decomposition of the time used iby different components is different: in Figure 6a, instruction fetch and ALU computation take a significant fraction of the time, while the percentage of time they use is much lower in Figure 6b. The main reason is that when the input size is larger, the cost of memory access is also larger. This means that optimizations reducing the number of memory accesses become more important as the input size becomes larger.

To further explore how different components and optimizations perform when we change the input size, we also show the average cost per cycle in Table 2. As expected, the average cost for instruction fetch and ALU computation does not depend on the input size, while the cost of memory access increases as the input size increases.

5.4 Performance of Binary Search

Binary search, where one party holds an array of integers while the other party holds a query, serves as an interesting test case for exploring the (amortized) speedup provided by working in the RAM model of computation. Secure RAM-based computation of binary search was considered by

	Average number of AND gates per cycle						
	Memory Access	Instruction Fetch	ALU Comp.	Total	Number of cycles	Total # of AND gates	Total time
TinyGarble	5423	2016	5316	12755	295	3.76M	–
Our system (baseline)	2067	9639	6723	18429	270	4.98M	12.8s
Our system (w/ optimizations)	1011	177	595	1783	270	481K	1.24s

Table 5: Comparison with TinyGarble [SHS+15] for computing Hamming distance on two integer arrays. Inputs are 32 32-bit integers.

Gordon et al. [GKK+12] and Wang et al. [WHC+14], both of whom used hand-crafted circuits. Liu et al. [LWN+15] also report an implementation of secure binary search, using techniques from Gentry et al. [GGH+13] and Wang et al. [WNL+14] which would not be available to non-expert users of their system.

For testing our system, we wrote C code for binary search using a standard iterative implementation (see Figure 11), with no annotations or special syntax, and then compiled this to a MIPS binary using GCC. In Figure 7, we show the performance of our system when emulating the resulting MIPS binary, when using ObliVM with Circuit ORAM [WCS15] or trivial ORAM as the back-end. In addition, we implemented secure binary search using ObliVM directly (without incorporating the optimization of Gentry et al. [GGH+13]), as well as via a "trivial" circuit that simply performs a linear scan. Since we only care about the amortized cost, setup time is excluded in all cases. As we can see in Figure 7, our emulator with Circuit ORAM outperforms the one with trivial ORAM once the array contains $> 2^{16}$ 32-bit integers, and outperforms a linear scan when the array contains $> 2^{19}$ 32-bit integers. We can also see that our emulator is only 25% slower than ObliVM, which requires the programmer to use the domain-specific language supported by ObliVM.

We further explored the effectiveness of instruction mapping in terms of number of memory accesses. In Table 4, we show the number of memory accesses in our emulator both before and after our instruction-mapping optimization. After we perform the optimization, the number of memory accesses decreases by an order of magnitude, approaching the number of accesses in the insecure setting.

5.5 Comparison with TinyGarble

Songhori et al. [SHS+15] also include a circuit for the MIPS architecture in their TinyGarble framework. Their MIPS circuit is based on existing hardware circuits, and assumes a memory bank and an instruction bank each containing exactly 64 32-bit integers. They use Hamming distance as an example to demonstrate the performance of their circuits, and provide a formula that can be used to compute the cost as a function of the input lengths.

The program used by TinyGarble does not follow the MIPS function-call convention, so we are not able to run their code directly on our system. However, we implemented the same functionality in C (see Figure 10), and compiled it to a MIPS binary using GCC. In Table 5, we show the

Size of the Tree	Memory Access	Instruction Fetch	ALU Computation	Number of Cycles	Total Time	Total Time for Circuit-based Approach
128	867 (1.3ms)	131 (0.2ms)	571 (1ms)	100	0.6s	0.1s
512	2694 (4.2ms)	135 (0.2ms)	578 (1ms)	120	1.1s	0.4s
2048	10979 (17.3ms)	137 (0.2ms)	587 (1ms)	150	3.3s	2.1s
8192	50225 (72ms)	139 (0.2ms)	591 (1ms)	180	13.8s	10.8s
32768	89961 (203ms)	140 (0.2ms)	595 (1ms)	210	43.8s	54.1s
65536	82307 (240ms)	141 (0.2ms)	597 (1ms)	220	53.9s	119.8s
131072	93616 (257ms)	141 (0.2ms)	598 (1ms)	240	62.8s	264.2s

Table 6: Number of AND gates and time for different components per cycle and total running time, for evaluating binary decision trees of different sizes.

performance of our system and TinyGarble. All numbers for TinyGarble are extracted from their online report [SHS+15]. Since the cost is related to the size of the memory and the program, we chose to use the maximum input size that their configuration can handle. Because the memory must hold input from both parties, the maximum input size TinyGarble can handle is 32 integers per party. For our system with and without optimization, we present the running time for the same input size.

We see from Table 5 that our baseline system performs worse than TinyGarble, even though they include the full MIPS instruction set in their ALU whereas we include only the set of instructions used in the binary code. What this demonstrates is that TinyGarble, through its use of optimization techniques from hardware design, produces much smaller circuits than ObliVM. As mentioned earlier, we could incorporate TinyGarble into our system—using it to produce the collection of garbled circuits, one for each time-step, that our system would execute—to improve performance of our system.

On the other hand, as we can see in Table 5, after we introduce our optimizations our system outperforms TinyGarble, with more than 7× improvement in the number of AND gates, a 5× decrease in the average number of AND gates per memory access, and about 9× speedup in the average number of AND gates for instruction fetching and ALU computation. This demonstrates the value of our optimizations. A system combining our techniques with the smaller circuits produced by TinyGarble would outperform either of those systems.

One interesting note is that the number of cycles used in our system, which is determined by the binary output by the GCC compiler, is slightly less than what is obtained from the hand-optimized code used by TinyGarble. This demonstrates the benefit of utilizing existing compilers.

5.6 Performance of Other Programs

Evaluating a binary decision tree. Consider a setting where one party holds a complete binary decision tree with n nodes, possibly padded with dummy nodes, and the other party holds an entry, namely e, of dimension m. They want to securely determine the label of the entry according to the binary decision tree. A circuit-based approach requires a circuit of size $O((m+n)\log(n))$. On the other hand, a RAM-based solution has an amortized cost of $O(\log n \log^2(m+n))$. As we will see in the following, in practice our RAM-based solution out-performs the best circuit-based solution we can come up with when the decision tree has more than $32K$ nodes.

Size of Decision Tree	128	512	2048	8192	32768	65536	131072
Number of Integers in Data bank	316	1084	4156	16444	65596	131132	262204
ORAM strategy	Trivial	Trivial	Trivial	Trivial	Circuit	Circuit	Circuit
#Mem access w/o optimization	100	120	150	180	210	220	240
#Mem access w/ optimization	21	26	32	39	45	47	51
Total Time spent in Mem access	0.13s	0.5s	2.59s	13.02s	42.7s	52.8s	61.66s

Table 7: Memory accesses for decision-tree evaluation.

Number of Nodes	40	50	60	70	80	90	100
Time for Instruction Fetch	$89.0s$	$133.8s$	$187.9s$	$246.5s$	$323.2s$	$401.0s$	$485.1s$
Time for ALU	$51.7s$	$76.9s$	$107.6s$	$142.1s$	$183.5s$	$227.9s$	$281.1s$
Time for Mem Access	$589.5s$	$1054.5s$	$1729.1s$	$2674.2s$	$3914.2s$	$5404.5s$	$7292.8s$
Run time	$737.9s$	$1276.4s$	$2040.4s$	$3083.5s$	$4447.6s$	$6066.5s$	$8099.5s$
Count	$14.8K$	$22K$	$30.6K$	$40.5K$	$51.9K$	$64.7K$	$78.9K$

Table 8: Performance of Dijkstra's Algorithm. Graphs with $E = 3V$ are used.

In a decision tree, each internal node is of the form $\langle val, idx \rangle$, which means that if $val \leq e[idx]$, we should branch left at this node; dummy nodes and leaf nodes are of the form $\langle lbl, -1 \rangle$, meaning that the return label associated with this path is lbl. Evaluation of a binary decision tree on entry e works by repeatedly evaluating the current node with e, and, depending on the result, branching left or right, until a dummy node or leaf node is reached.

In this experiment, we fix the dimension of an entry as 20 and vary the size of the binary decision tree. We implemented the above algorithm directly in C (code can be found in the Appendix), which is then used to generate the MIPS binary file. As we can see from Table 6, it takes 0.5 second to evaluate a binary decision tree with 64 nodes, and around 1 minute to evaluate such a tree with about $131K$ nodes. We also implemented a circuit based solution with our best effort to minimize the circuit size. When comparing our RAM-based solution with this circuit-based approach, we find that it performs better than circuit-based solution for decision trees with more than 32768 nodes.

The table also indicates that the cost is dominated by the cost of the oblivious memory access when the size of the binary decision tree is large. For example, 99% of the time is spent on oblivious memory access for a decision tree with 2048 nodes. Therefore we look into the cost of memory accesses more carefully in Table 7. Similar to the binary search example, we start using Circuit ORAM when the size is larger than 2^{16}, which corresponds to a decision tree with $32K$ nodes. We find that our optimizations reduce the number of memory accesses by $5\times$.

Dijkstra's Algorithm.

Dijkstra's algorithm has been studied in several previous works [LWN+15, LHS+14, KS14, ACM+13]. They mainly focused on how to design customized cryptographic protocol or oblivious algorithm to improve the running time. ObliVM [LWN+15] proposed a specific programming pattern that helps programmers to write better versions of the algorithm, but programmers still need to understand the concept of obliviousness as well as special syntax and annotations. We

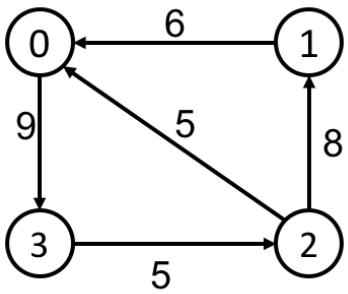


Figure 8: A simple graph with 4 nodes and 5 edges.

implemented Dijkstra's algorithm on general graphs using C which is compiled to MIPS binary for our use. The code is provided in the Appendix.

Similar to some previous work [LWN+15, KS14], we reveal the number of nodes and edges for better efficiency. The graph is organized as an adjacency list. In particular, one party holds two arrays `edge` and `weight`, containing the tail node and weight of each edge respectively, and sorted according to the head node of the edge. This party also holds an array `node` where `node[i]` is the sum of out degree of node `1` to `i-1`. This also means that all edges started from node `i` has tail nodes `edge[node[i]], edge[node[i]+1], ..., edge[node[i+1]]-1`. For example, a simple graph shown in Figure 8 will be represented as:

```
node   = {0, 1, 2, 4}
edge   = {3, 0, 0, 1, 2}
weight = {9, 6, 5 ,8, 5}.
```

We ran Dijkstra's algorithm on sparse graphs with $E = 3V$ and summarized the results in Table 8. We can see from the table that it takes about 20 minutes for us to run Dijkstra's algorithm on a graph with 50 nodes. Keller et al. implemented Dijkstra on sparse graphs. Their implementation is based on SPDZ protocol and it takes them about 100 seconds online time for a graph of 100 nodes. Our online time is about 80 times slower than theirs. If we consider the total time, then SPDZ has the longer running time due to its tremendous offline stage, which is roughly 200 times slower than its online time.

6 Conclusion

In this work, we designed and implemented an emulator that allows two parties to securely execute native MIPS code. Our work fills an important gap in the trade-off between efficiency and generality: by supporting MIPS code, we allow programmers with no knowledge or understanding of secure computation to write and securely execute code of their choice. Contrary to what one might expect, we show that this approach can yield reasonable performance once several automated optimizations are applied. For some programs, our optimized system is competitive with implementations based on state-of-the-art domain-specific languages for secure computation.

There are still many interesting, unexplored optimizations that would further improve the efficiency of our approach: (1) In this work, we demonstrate the potential advantages of padding with NOP instruction, but it remains an interesting challenge to automate and optimize this step. (2) It is also interesting to explore how taint analysis, and other more complex analysis of binary

files, can improve the performance by allowing us to further avoid oblivious memory accesses and unnecessary secure computation in the ALU. Our work demonstrates the feasibility of the most general approach to secure computation, opening an avenue for further research and helping to fill a gap in the growing array of options for performing secure computation.

Acknowledgements

The authors thank Elaine Shi for helpful discussions in the early stages. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). Work of Xiao Wang and Jonathan Katz was additionally supported in part by NSF awards #1111599 and #1563722. The views, opinions, and/or findings contained in this work are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[ACM+13] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. Securely solving simple combinatorial graph problems. In *Financial Cryptography and Data Security*, volume 7859 of *LNCS*, pages 239–257. Springer, 2013.

[AMPR14] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. Non-interactive secure computation based on cut-and-choose. In *Advances in Cryptology–EUROCRYPT 2014*, pages 387–404. Springer, 2014.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, October 2008.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY—A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[EFLL12] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: The secure computation application programming interface. Cryptology ePrint Archive, Report 2012/629, 2012.

[FDD12] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. Towards an interpreter for efficient encrypted computation. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, 2012.

[GGH+13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS*, 2013.

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

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Usenix Security Symposium*, 2011.

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS*, pages 772–783. ACM Press, October 2012.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS*, pages 451–462. ACM Press, October 2010.

[Kel15] Marcel Keller. The oblivious machine - or: How to put the c into mpc. Cryptology ePrint Archive, Report 2015/467, 2015. http://eprint.iacr.org/.

[KMSB13] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Usenix Security Symposium*, 2013.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *Asiacrypt*, 2014.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, 2012.

[LHS+14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient RAM-model secure computation. In *IEEE Security & Privacy*, 2014.

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In *ACM CCS 15*, pages 579–590. ACM Press, 2015.

[LWN+15] Chang Liu, Xiao Shaun Wang, Karthik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *IEEE Security & Privacy*, 2015.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay: a secure two-party computation system. In *USENIX Security Symposium*, 2004.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Asiacrypt*, volume 5912 of *LNCS*, pages 250–267. Springer, December 2009.

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670. IEEE Computer Society Press, May 2014.

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE Security & Privacy*, 2015.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *ACM CCS 15*, pages 850–861. ACM Press, 2015.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS*, pages 191–202. ACM Press, November 2014.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS*, pages 215–226. ACM Press, November 2014.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[ZE15] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015.

Computing the size of set intersection

```
int intersection(int *a, int *b,
                 int l, int l2) {
  int i = 0, j = 0, total=0;
  while(i < l && j < l2) {
    if(a[i] < b[j])
      i++;
    else if(b[j] < a[i])
      j++;
    else {
      i++;
      total++;
    }
  }
  return total;
}
```

Figure 9: Code for computing the size of the intersection of two sets.

C code for Hamming distance

```
int hamming(int *a, int *b, int l) {
   int *e = a + l;
   l = 0;
   while(a < e) {
      if(*a++ != *b++)
         ++l;
   }
 return l;
}
```

Figure 10: C code for Hamming distance

C code for binary search

```
int binarySearch(int *a, int *b, int l) {
   int key = b[0], imin = 0, imax = l-1;
   while (imax >= imin) {
      int imid = (imin+ imax)/2;
      if (a[imid] >= key)
         imax = imid - 1;
      else
         imin = imid + 1;
   }
   return imin;
}
```

Figure 11: C code for binary search

C code for binary decision-tree evaluation

```
int binaryDecisionTree(int *a, int *b) {
  int depth = a[0];
  int * tree = a+1;
  int index = 0;
  for(int i = 1; i < depth; ++i) {
    if(tree[index*2+1] != -1) {
      int rbranch =
       tree[index*2]<=b[tree[index*2+1]];
      index = 2*index+1;
      index += right_branch;
     }
  }
  return tree[index*2];
}
```

Figure 12: C code for binary decision-tree evaluation

C code for Dijkstra's algorithm

```
#define MAX 100
#define MAX_INT (1<<31)-1
int dijkstra(int *a, int *b){
 int vis[MAX];
 int dis[MAX];
 dis[b[0]] = 0;
 int n = a[0];
 int e = a[1];
 int * node = a + 2;
 int * edge = a + 2 + n;
 int * weight = a + 2 + n + e;
 for(int i = 0; i < n; ++i) {
  int bestj = -1, bestdis = MAX_INT;
  for(int j=0; j<n; ++j) {
   if( (!vis[j]) & dis[j] < bestdis) {
    bestj = j;
    bestdis = dis[j];
    }
   }
   vis[bestj] = 1;
   for(int j = node[bestj];
     j <= node[bestj+1]; ++j) {
    int newDis = bestdis + weight[j];
    if(newDis < dis[edge[j]])
     dis[edge[j]] = newDis;
   }
 }
 return dis[b[1]];
}
```

Figure 13: C code for Dijkstra's algorithm