Secure Computation of MIPS Machine Code

Xiao Shaun Wang* S. Dov Gordon† Allen McIntosh†
Jonathan Katz*

Abstract

Existing systems for secure computation require programmers to express the program to be securely computed as a *circuit*, or in some *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 has the advantage of allowing 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 existing "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 ALU 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 some agreed-upon program on their inputs while revealing nothing to either party other than the result of the computation. This notion originated in the cryptography community in the early 1980s [19], and until relatively 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 [13], 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 [10, 14, 5, 7, 9, 6, 11, 8, 15, 12, 16].

Many (though not all) of these implementations actually 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, the end-user is forced to express their program in a particular, domain-specific language [6, 20, 11, 12, 15] that can be hard to learn and use, or is simply limited in terms of expressiveness. Even if the language is a subset of a standard language (such as ANSI C, as in [6]), the programmer must still be aware of which features of the language to avoid; moreover, legacy code will not be supported, and the fact remains that the user cannot write the program in a language of their choice. A second drawback is that compilers from high-level languages to boolean circuits (or other suitable representations) can be slow; for example, in our own experiments we found that the CBMC-GC compiler [6] took *10 days* to compile a short program, which performs a linear scan over a 2000-element array, into a circuit.

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 – 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

*Department of Computer Science, University of Maryland. Email: {wangxiao,jkatz}@cs.umd.edu
†Applied Communication Sciences. Email: {sgordon,amcintosh}@appcomsci.com

uses oblivious RAM [3] 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 each of the problems mentioned earlier:

- Programmers can write their programs in any language of their choice, and compile them using any compiler of their choice, so long as they end up with MIPS machine code.
- 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 available at `https://github.com/wangxiao1254/Secure-Computation-of-MIPS-Machine-Code`.

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 [4, 11, 12]. 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[1] 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 [4] (though only in an amortized sense); we show an example of this in Section 5.4.

As an independent advantage, our system seamlessly supports *private function evaluation*, in which one party's private input is itself a program. The parties simply compile and execute a program that takes a pointer to the private function as one input, the other party's private value as a second input, and calls the former on the latter. The private function can be written in any language, and the off-the-shelf compiler then takes care of the rest.[2] Although private function evaluation has been implemented before, to our knowledge,all prior work requires the programmer to express the private function as a boolean circuit.

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 and that can be done before the parties' inputs are known and before execution of any protocol between them begins. These optimizations improve the performance of our system to the point where it is feasible; in fact, for certain applications we are only $\sim$25% slower than ObliVM [12] (see Section 5.4).

Why MIPS? As noted above, our goal was to design a system for secure computation of programs expressed in a low-level representation. To this end, we examined various programming architectures and determined that MIPS was the most suitable choice for our purposes. We briefly explain our decision.

We considered using the LLVM intermediate representation, but this was deemed too complex. In particular, LLVM supports function calls as an atomic abstraction, which are hard to incorporate directly into a (garbled) circuit while preserving efficiency. Considering this, we decided to use a lower-level language to simplify the design.

In general, the secure computation of a program requires hiding which instruction is being executed at a given time step. This means that each instruction emulated will incur (at least) the worst-case cost among all instructions that could potentially be executed at that time step. For this reason, we decided to work with a Reduced Instruction Set Computing (RISC) architecture rather than a Complex Instruction Set Computing (CISC) architecture. Even some RISC architectures are too complex for our purposes. For example, ARM

[1] Note that here we allow for *input-dependent running time*, which can in general leak information. See the discussion in the following section.

[2] In this scenario, we cannot perform the binary analysis that leads to the optimizations described in Section 4, but the performance reported from our baseline construction of Section 3 would apply directly, without further overhead.

supports variable-length instructions, allows many instructions to be executed conditionally, and enables optional shifting of register values before use. This certainly complicates the design of a garbled circuit for CPU execution and would likely hurt the efficiency of secure computation based on ARM.

That said, a more complex architecture would reduce the number of cycles required in the computation, so there is certainly a trade-off that could be explored. Although we focus on MIPS, the ideas we describe are generally applicable and we see no fundamental reason why our work could not be adapted to other architectures.

Comparison to TinyGarble. In concurrent and independent work, Songhori et al. [16] explored using hardware-optimization techniques to reduce the size of boolean circuits for common functions, with applications to secure computation in mind. As part of their work, they developed a circuit for evaluating a MIPS CPU. However, the goal of their work was different, and they do not build a full-fledged system for executing native MIPS code; in particular, their circuit assumes fixed size memory and does not use oblivious RAM for oblivious memory access. Furthermore, they do not investigate optimizations to accelerate execution of the program, something that is a key contribution of our work. Their work is essentially orthogonal to ours; by using their circuits in our emulator, we could build a system that out-performs either of the original systems.

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.

Our emulator makes extensive use of Yao's garbled-circuit protocol [19] 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 [3] allows a "client" to read/write an array of data stored on a "server," while hiding from the server both the content of the data as well as the data-access pattern. The definition of ORAM only hides information from the server; to hide information from both parties, we adopt the method of Gordon et al. [4] 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' secret-shares of the state.

In this work, we use ORAM to store both the MIPS program (so parties remain oblivious about the instruction being fetched) as well as the contents of main memory (so parties remain 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 an ORAM storing MIPS instructions as an "instruction bank," and an ORAM storing memory contents as a "memory bank."

ObliVM. Our system is based on an emulator that securely computes each cycle of a MIPS CPU. We use ObliVM [12] to generate the code for this emulator. ObliVM supports ORAM (for oblivious memory accesses) as well as garbled circuits (for secure computation) using 80-bit keys. For the ORAM component, ObliVM determines whether to use circuit ORAM [17] or "trivial ORAM" (i.e., a multiplexer over the entire array) based on the array length (an upper bound on which must be known in advance).

Security of our emulator. Although we do not provide any proofs of security in this work, an argument as in [4] can be used to prove that our construction is secure (in the semi-honest model) with respect to standard definitions of security [2], with one important caveat: our system currently 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 forbidden in prior work. In some cases, 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 other cases, however, one can show that leaking the running time leaks no information; an example of this is when running a randomized algorithm whose running time depends on the randomness, but not the private inputs. In fact, in this latter case our system would have the advantage of having an expected running time proportional to the expected running time of the original algorithm; in contrast, prior work would (at best) incur a running time proportional to the *worst-case* running time, and in most cases, would simply fail to compile.

3 Basic System Design

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

3.1 MIPS Architecture

A MIPS program is expressed as an array of instructions, each exactly 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. Each instruction type can take different registers as arguments and therefore there are many possible instructions of each type.

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 at the location specified by the program counter.
- Instruction decode (`ID`): fetch 2 registers, whose indices depend on the instruction.
- Execute (`EX`): execute the instruction and update the program counter.
- Memory access (`MEM`): perform load/store operations on memory, if applicable (depending on the instruction).
- Write back (`WB`): write a value to one register.

3.2 Overview of Our System

At a high level, the two parties in our system 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.

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. However, since there are only 32 registers, a trivial ORAM (i.e., just 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 over 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 in order 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 (this corresponds to 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 both steps 3 and 4 above in every cycle, even if the step ends up having no effect on the shared MIPS state. (Our improved design in Section 4 provides a way of securely bypassing step 4 – the memory component – on many cycles.)

Details of the above are given in the following sections.

3.3 Setup

Before we start to execute the main loop emulating an instruction cycle, we need to load the MIPS code into the instruction memory and the users' inputs into the main memory.

Loading the MIPS code. In our baseline system design, we load the full program into a single ORAM. Therefore, at each step, we incur the cost of accessing a large ORAM containing instructions from the whole program. In Section 4 we will describe improvements to this approach.

In our current implementation, we do not load any code that is executed before `main()` is called, such as library start-up code, code for managing dynamic libraries, and 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 so could be simulated by the parties locally. Adding support for loading such code would thus be easy to incorporate in our system.

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. The parties reconstruct the output contained in reg or $\mathsf{ORAM}_{\mathsf{MEM}}$.

Figure 1: Overview of secure computation of a MIPS program. Boxed lines are executed outside of secure computation.

Loading user inputs. We assume a public upper bound on the input size of each party. Each party places their input in a local file. When the emulation begins, the two parties initialize an empty ORAM supporting the maximum input size, and the parties' input values are secret shared and written to some agreed-upon (non-overlapping) segments of memory. The two parties also initialize their shares of the register space with the pertinent information, such as the address and length of the input data.

3.4 Main Execution Loop

We use $\mathsf{ORAM}_{\mathsf{inst}}[], \mathsf{ORAM}_{\mathsf{MEM}}[]$, reg[], pc and inst to denote, respectively, the (shared) instruction ORAM bank, (shared) memory ORAM bank, (shared) registers, (shared) value of the program counter, and (shared) current instruction. As shown in Figure 1, secure execution of a MIPS program involves repeated calls of three procedures: instruction fetch, ALU computation, and memory access. We describe these below.

Instruction fetch. In the basic system, we put the entire MIPS program into a single ORAM. Therefore, fetching the next instruction is simply a secure two-party ORAM access.

ALU computation. The MIPS ALU is securely computed using a universal garbled circuit. As shown in Figure 2, 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 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.

Among these steps, 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.

The circuit size for the third step depends on the set of instruction types supported by the universal circuit emulating the ALU. In our basic system design, we support about 30 instruction types (see Table 1) using a circuit containing 6700 AND gates. We found that these instructions sufficed for all the MIPS programs we used in our experiments: set intersection, binary search and Hamming distance. They also suffice for many other programs commonly used in secure computation, such as longest common sub-string, heapsort and Djikstra's algorithm, to name a few. Users of our system can easily extend the set of supported instructions as the need arises; each one requires two or three lines of code, written in the ObliVM language. As we will describe in Section 4, our system can automatically build an ALU containing only the instruction

Secure computation of the MIPS ALU

Input: inst, pc, reg[]

ALU:

1. rs := inst[21:25], funct := inst[0:5],
   rd := inst[11:15], op := inst[26:31],
   rt := inst[16:20], 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
       pc := pc + 4
   else if op == I_Type and funct == BNE
       pc := (reg_rs == reg_rt)?(pc + 4 + 4 × imm) : pc
   else if . . . //more cases that update pc
5. if op == R_Type
       index := rd, value := reg_rd
   else
       index := rt, value := reg_rt
   reg[index] := value

Figure 2: This functionality takes the program counter, current instruction, and registers as input. Depending on the type of the instruction, it performs computation and updates the register space, as well as updating the program counter.

types that are contained in the current program. Therefore, adding more instructions to support a new program will not impact the performance of other programs.

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 3, 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 this component 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 should be terminated. 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. See Figure 4.

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 architecture 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 an expensive accesses to the memory ORAM in every instruction cycle.

In Section 4.1 we describe how we use static analysis on the MIPS code to assign to every time-step of the execution a subset of the program instructions that can possibly be executed at that time-step. The

Secure computation of memory load/store

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

MEM:

1. rs := inst[21:25], op := inst[26:31],
   rt := inst[16:20], immediate := inst[0:15]
2. addr := immediate+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]

Figure 3: This functionality takes the current instruction, the registers, and the memory as input. Depending on the instruction type, it reads a value from memory to a register, or writes a value from a register to memory.

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. The parties reconstruct the bit terminate.

Figure 4: This functionality takes the current instruction and the registers as input. It returns true if these indicate program termination.

result is that, in a given time-step, we only need to evaluate a universal circuit over the subset of possible instructions at that time-step; as we will see, this allows us to dramatically improve the run-time of our system in almost every instruction cycle. In Section 4.2, we show that performance can be further improved by adding empty (NOP) instructions in carefully chosen places of the code. Although such NOPs increase the number of time-steps, they can (for certain programs) reduce the number of possible instructions per time-step, leading to a net decrease in the running time.

In Section 4.3, we highlight an independent optimization that reduces the number of communication rounds in the protocol, and also leads to a significant improvement in the running time.

4.1 Mapping Instructions to Time-steps

We improve the efficiency of our system by identifying unnecessary computation. First, we perform static analysis on the MIPS binary code and compute, for every time-step of the program execution, a set of the 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 the following ways:

1. When fetching instructions, we only need to search through the instruction bank that is assigned to the current time-step, rather than the entire program. This improves the performance of the `IF` stage.
2. When performing ALU computation, we only need to incorporate instruction types that appear in the current instruction bank. This improves the performance of the `EX` stage.

R Type	ADDU	SUBU	SLL	SLTU
	SRL	SRA	AND	OR
	XOR	NOR	MOVZ	MOVN
I Type	ADDIU	XORI	ANDI	ORI
	SLTIU	BNE	BEQ	LUI
	BLEZ	BGTZ	BGEZ	BLTZ
	BLTZAL	BGEZAL	JR	
J Type	J	JAL		

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

Time-step assignments: Example 1

```
main:
   MULT $t1, $t2, $t3      //time-step 1
   BNE $t1, $t2, else      //time-step 2
   Instruction 1           //time-step 3
   Instruction 2           //time-step 4
   J endif                 //time-step 5
else:
   Instruction 3           //time-step 3
   Instruction 4           //time-step 4
   Instruction 5           //time-step 5
endif:
```

Figure 5: An example demonstrating how we assign instructions to time-steps in order to avoid executing the full architecture on each cycle. The multiplication in time-step 1 will be excluded from the ALU circuit after the first time-step.

3. If the current instruction bank does not contain a `load` or `store` instruction, we do not have to read or write the memory ORAM in this time-step. This improves the performance of the `MEM` stage.
4. We can access a reduced set of registers by accessing only those that appear in the instructions mapped to the current time-step. This improves the performance of the `EX` and `MEM` stages.

Computing instruction banks for each time-step. To map each time-step to a set of instructions that can be executed at that time-step, we step through the binary, spawning a new thread each time we encounter a branching instruction. Each thread moves forward, tagging the instructions it encounters with an increasing time-step, and spawning again when it comes upon a branch. We terminate our analysis if all threads halt, or if we see that the set of instructions tagged in some time-step (i.e., across all threads) has repeated itself. It is easy to verify that one of these two conditions will eventually be met. In the worst case, the analysis will terminate when every instruction is assigned to a single time-step.

To illustrate this, we provide a very simple example in Figure 5. Although there are seven instructions in the MIPS code snippet, at most two instructions can possibly be executed in any time-step. When the code contains loops, as shown in Figure 6, 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.

This analysis does not result in an optimal assignment of instructions to time-steps, because it ignores data values. In particular, we treat every branch as though it were "tainted" by private inputs, even if the program branches on a public value. When this occurs, note that both parties could conceivably compute which side of the branch will be executed, and we could include only the relevant branch in our instruction

Time-step assignments: Example 2

```
main:
   ADD $t1, $t2, $t3     //time-step 1
while:
   Instruction 1         //time-step 2, k+3,
   ...
   Instruction k         //time-step k+1, 2k+2,
   BNE $t1, $t2, while   //time-step k+2, 2k+3,
post-while:
   Instruction k+1       //time-step k+3, 2k+4
   Instruction k+2       //time-step k+4, 2k+5
   ...
   Instruction 2k+1      //time-step 2k+3, 3k+4
```

Figure 6: An example demonstrating how we map instructions to time-steps with loops in the program.

set. To some extent, the compiler takes care of this for us through constant folding and loop unrolling, but there are scenarios in which it will fail to do that (such as if we are compiling from multiple source files). We leave it to future work to explore better methods of performing this mapping through taint analysis and constant folding. For branches that do depend on tainted data, our analysis is optimal; in this case, in order to maintain privacy, all instructions that we map to a time-step *must* be in the ALU for that time-step, or we would leak information about the user inputs. Conversely, because our analysis is data-independent, it is easy to see that it does not leak any private information. As a final comment, it is interesting to note that we cannot perform this analysis if we wish to support dynamic libraries. In this case, we are stuck with the base-line performance described in the previous section.

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 first find the instruction bank corresponding to the current time-step, and then perform an oblivious fetch on this small instruction bank.

An interesting observation is that when we employ this optimization, using an ORAM to store a set of instructions becomes inefficient. Originally, instructions were located continuously, so N instructions could be put into an ORAM of size N. Now, each instruction bank contains only a small subset of instructions, say of size $x < N$, while their address values still span the original range. If we use an ORAM to store them, its size would have to be N instead of x. Therefore, we use an oblivious key-value structure to store a set of instructions for each time-step. Since the size of each instruction bank is very small for the programs we tested, we only implemented an oblivious key-value structure that supports a linear scan; we leave it to future work to support this optimization in programs with a large enough state-space to benefit from sub-linear access time for instruction fetching.

Constructing smaller ALU circuits. Once we have reduced the set of plausible instructions in a given time-step, we can also reduce the set of *instruction types* required in that time-step. Before user inputs are specified, we generate a distinct ALU circuit for each time-step, using the language from ObliVM, and compiling them with the ObliVM compiler. During run-time, our emulator will publicly choose, according to the current time-step, the appropriate ALU circuit to run.

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 dynamically skips the load/store computation depending on the values of these flags.

Improving accesses to the register space. We additionally performed an optimization that further improves the speed of register accesses. Note that the register values are hard-coded into the instructions, and 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.

At every time-step, we compute a set of register addresses that covers all possible register accesses at this time-step. For example, if at a certain time-step, the instruction bank contains only `ADD $4, $1, $2` and `ADD $5, $1, $3`, then we just need to touch register 1 when loading reg[rs], touch register 2 and 3 when loading reg[rt] and touch registers 4 and 5 when accessing reg[rd].

During the static analysis, we compute a mask for each time-step indicating which registers must be touched in the ALU computation and memory operations for that time-step. Only those registers are included in the secure emulation for that time-step.

4.2 Padding Branches

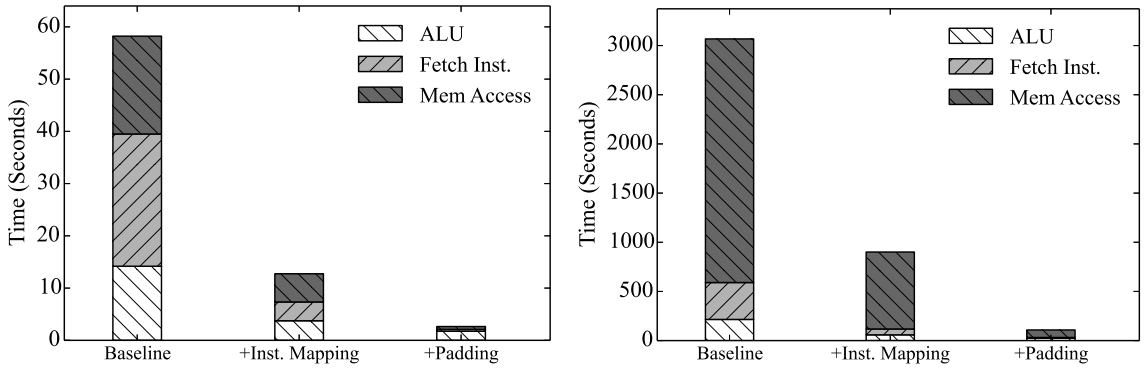
The offline analysis we just described provides substantial improvements, on real programs. For example, as we show in Section 5.2, for the case of set intersection, this analysis reduces the cost of instruction fetch by $6\times$ and reduces the ALU circuit size by $1.5\times$. This is because the full program has about 150 instructions, while the largest instruction bank after performing our binary analysis contains just 31 instructions, and for more than half of our program execution, we fetch instructions from banks containing less than 20 items. On the other hand, there is only a small reduction in the time spent on performing load and store operations, because these operations are plausible in nearly every time-step. Load and store operations are by far the most costly instructions performed in these programs. For example, reading a value from a 1024-sized memory 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 was already described. Consider again the simple example in Figure 6. Here, with only a single loop and no branches inside of it, 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—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 of this loop, resulting in two parallel basic blocks of length 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 less than $k_1 \cdot k_2$ time-steps, we could be anywhere in the loop! Furthermore, at that point we might exit the loop on any step, so after less 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 leads to a natural idea of padding parallel blocks so that the length of one is a multiple of the other. Using the same example of a single if/else statement inside of a loop, if the two branches have equal length, $k_1 = k_2$, 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 bank.

To take a real world example, we used the code for set intersection and padded 2 of the three branches that appear in the main loop, using a total of 6 nop instructions. The maximum number of instructions in each bank goes from 30 to just 4 after we pad appropriately and re-run our binary analysis. Even more importantly, 1700 out of 1750 time-steps have a load/store operation before we add the padding. Afterward, these operations are executed only 192 times out of 2026. We provide further analysis in Section 5.



(a) Running time decomposition for set intersection when each party's input consists of 64 32-bit integers.

(b) Running time decomposition for set intersection when each party's input consists of 1024 32-bit integers.

Our results demonstrate that padding parallel branches can offer tremendous savings, when combined with optimizations previously discussed. It is a very interesting question to automate this process so that it can be leveraged in programs with more complex structure than set intersection.

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. We found this overhead to be significant, especially after we performed our optimizations mentioned in Sections 4.1 and 4.2. For example, for set intersection with inputs of size of 1024, our optimized system requires about 5 milliseconds to emulate each cycle, while the time for round-trip communication is much higher.

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 immediately. All memory and register accesses now take this bit as input, and do not update the memory or registers if the program has terminated. This ensures that the MIPS state will not change after termination, even if more 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.

5 Performance Analysis

5.1 Experimental Setup

Our emulator takes MIPS binaries as input and executes the binary code securely between two parties. All binaries we used were generated from C code with an off-the-shelf GCC compiler compiling C programs to MIPS binaries. When specified, we added NOPs to the binary by hand to further explore the potential speedup introduced by padding; for other examples, including Hamming distance and binary search, we directly used the binaries generated by the GCC compiler with no modification.

All times reported are based on two machines running on the Amazon EC2 cloud service. Each machine is of type `c4.4xlarge`, with 3.3GHz cores and 30GB memory. The two parties communicate through a Local Area Network (LAN) within the same region of the Amazon EC2 cluster. We did not use parallel cores, except possibly by the JVM for garbage collection. In all reported running times, we report the time taken for the generator, which is the bottleneck in the garbled-circuit protocol.

Input size / party		Memory Access	Instruction Fetch	ALU Computation	Average / 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 the size of a set intersection.

Metrics. Since our emulator uses ALU circuits and instruction banks with different sizes for 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.,

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

– 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 The Effect of Our Optimizations

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

The program used here 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 8. In Figure 7a and Figure 7b, we show the decomposition of running time when the input from each party is 64 and 1024 integers. We do not include the setup time in the figure because it is too small to be visible against the time for the main loop. However, we report the exact running time, including the setup time, in Table 5 in the Appendix. Note that for all running times shown, including the baseline, we incorporated the optimization that checks termination less frequently, as described in Section 4.3.

In summary, our optimized system achieves about 28× speedup compared to the basic design described in Section 3. It reduces the cost of instruction fetch by 67×, reduces the cost of memory access by 33×, and reduces the cost of ALU Computation by 8×.

In the following, we also detail the source of the speedup at different stages of optimizations.

– **Baseline System.** A basic system described previously with no static analysis. (Section 3)

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

Table 3: Number of memory accesses for binary search with different size inputs.

	Average number of AND gates per cycle						
	Memory Access	Instruction Fetch	ALU Comp.	AND gates per cycle	Number of cycles	Total # of AND gates	Total time
TinyGarble (32)	5423	2016	5316	12755	295	3762K	–
Our System (32)	1011	177	595	1783	270	481K	1.24s
Our System (512)	11206	180	601	11987	4140	49.6M	75.1s

Table 4: Comparison with TinyGarble [16] for computing Hamming distance on two integer arrays. The length of the arrays is indicated in parentheses.

– **+Instruction Mapping** For each time step the compiler automatically computes a set of instructions that can possibly be run, and uses a smaller instruction bank and ALU circuit for each cycle, reduces register accesses, and bypasses load/stores whenever possible, as described previously. (Section 4.1)

  As we can see from Table 5, by introducing this optimization, the time used in memory access is decreased by 3×, the cost of instruction fetch is reduced by 6×, and the ALU computation is reduced by 3.5×.

– **+Padding.** Dummy instructions are added to pad instructions in inner loops. (Section 4.2)

  After padding, we can see a further 10× improvement on both instruction fetch and memory accesses. There is also a 2× speedup in ALU computation.

If we compare Figure 7a where the input size is 64 integers and Figure 7b where input size is 1024 integers, we find that the decomposition of the time used in different components is different: in Figure 7a, instruction fetch and ALU computation take a significant fraction of the time, while the percentage of time they use is much lower in Figure 7b. The main reason is that when the input size is larger, the cost of memory access is also larger. This means that optimizations on 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.

In Table 3, we show the number of memory accesses in our emulator both before and after the instruction mapping optimization described in Section 4.1. As we can see from the table, 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.

5.3 Comparison with TinyGarble

Songhori et al. [16] also include a circuit for the MIPS architecture in their TinyGarble framework. Their MIPS circuit is based on existing hardware projects, assuming a memory and an instruction bank each containing only 64 32-bit integers. They use Hamming distance as an example to show the performance of their circuits, with a formula that can be used to compute the cost.

The program used in 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 and compiled

it to a MIPS binary. The C code used to generate the MIPS binary is shown in Figure 9. In Table 4, we show the performance of our system and TinyGarble. All numbers from TinyGarble are extracted from their online report [16]. Since the cost is related to the size of the memory and the program, we chose to use the maximum input size that TinyGarble can handle. Because the memory needs to hold input from both parties, the maximum input size TinyGarble can handle is 32 integers per party. For our system, we present the cost with input sizes 32 and 512, with wall-clock running time, including setup cost.

Since the MIPS circuit generated by TinyGarble is from an existing hardware project, their circuit is very compact and likely smaller than the ones we generate using ObliVM. However, our system still outperforms TinyGarble, due to our optimizations. As we can see in Table 4, our system achieves an overall $7\times$ improvement compared to TinyGarble's MIPS circuits. (Recall, are ALU includes only the instruction types required in its time-step, while they include the full architecture.) There is a $5\times$ speedup in the average number of AND gates per memory access, and about $9\times$ speedup in the average number of AND gates for instruction fetch and ALU computation. We can also see from the table that even after we use inputs of size 512, the cost per cycle is still less than the cost of executing TinyGarble with inputs of size 32. This illustrates the effectiveness and importance of our optimizations based on static analysis of the code.

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

5.4 Performance of Binary Search

We also wrote C code for binary search with a standard iterative implementation. In our setting, one of the parties holds an array of integers while the other party holds a query. This functionality has been implemented by several previous works: Gordon et al. [4] and Wang et al. [18] implemented binary search with hand-built circuits, which is impossible for non-expert programmers that have no domain knowledge. ObliVM also reports a binary search implementation, but this incorporates techniques from Gentry et al. [1] and requires knowledge on the construction of an ORAM. In contrast, our binary search implementation is solely written in C code as shown in Figure 10, with no annotation or special syntax.

In Figure 7, we show the performance of our emulator with Circuit ORAM and trivial ORAM. In addition, we also wrote a binary search in ObliVM without performing the advanced optimization introduced by Gentry et al. [1]. All those numbers are compared with a circuit that linearly scans through the array. Since we only care about the amortized cost, setup time is excluded in all cases. As we can see from the Figure 7, our emulator with Circuit ORAM outperforms the one with trivial ORAM when the array contains 2^{16} integers, and outperforms a linear scan circuit when the array contains more than 2^{19} integers. We can also see that our emulator is only 25% slower than ObliVM, which relies on special annotations and syntax.

6 Conclusion

In this work, we designed and implemented an emulator that takes MIPS binary files generated by an off-the-shelf compiler and performs a secure two-party computation of the binary. Our work fills an important gap in the trade-off between efficiency and generality: by supporting legacy code, we allow programmers with no knowledge or understanding of secure computation to write and execute code for that task. Contrary to the expectations of many researchers in the field, we show that the approach of using a "Universal Circuit" to securely execute RAM model programs can yield reasonable performance. Our optimized protocol achieves significant speedup compared with a baseline system and, for some programs, is even competitive with an implementation using a state-of-the-art domain-specific language.

As mentioned in previous sections, 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 this task. 2) It is also interesting to explore how taint analysis, and other more complex analysis of binary files will improve the performance by allowing us to further avoid oblivious memory accesses and unnecessary secure computation

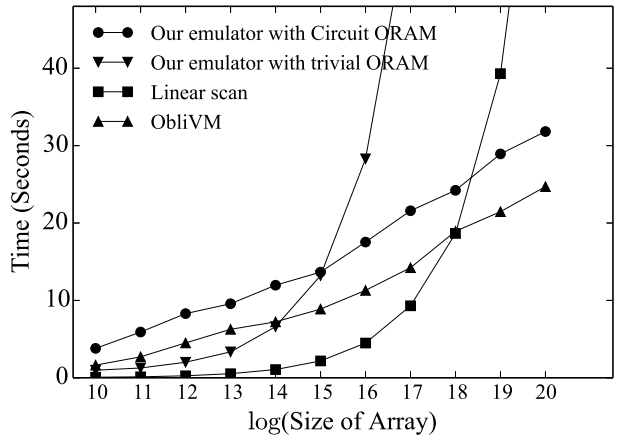


Figure 7: **Comparing the performance of binary search with and without ORAM.** One party holds an array of 32-bit integers, while the other holds a value to search for.

in the ALU. Our work demonstrates the feasibility of the most general approach to secure computation, opening an important 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 of this work. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). Work of Jonathan Katz was additionally supported by NSF award #1111599. 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

[1] 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.

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

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

[4] 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 *ACM CCS 12*, pages 513–524, 2012.

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

[6] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In *CCS*, 2012.

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

[8] 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.

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

[10] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN 2008*, volume 5229 of *LNCS*, pages 2–20. Springer, 2008.

[11] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating Efficient RAM-model Secure Computation. In *S & P*, May 2014.

[12] Chang Liu, Xiao Shaun Wang, Karthik Nayak, Yan Huang, and Elaine Shi. ObliVM: A generic, customizable, and reusable secure computation architecture. In *IEEE S & P*, 2015.

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

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

[15] 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, 2014.

[16] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *IEEE S & P*, 2015.

[17] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. Cryptology ePrint Archive, Report 2014/672, 2014. `http://eprint.iacr.org/`.

[18] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *CCS*, 2014.

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

[20] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In *CCS*, 2013.

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

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 8: 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 l2) {
   int *e = a + l;
   l = 0;
   while(a < e) {
      if(*a++ == *b++)
         ++l;
   }
 return l;
}
```

Figure 9: C code for Hamming distance

C code for binary search

```
int binarySearch(int *a, int *b, int l, int l2) {
   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 10: C code for binary search