

Secure Computation of MIPS Machine Code

Xiao Wang¹(✉), S. Dov Gordon², Allen McIntosh³, and Jonathan Katz¹

¹ University of Maryland, College Park, USA

{wangxiao, jkatz}@cs.umd.edu

² George Mason University, Fairfax, USA

gordon@gmu.edu

³ Applied Communication Sciences, Basking Ridge, USA

amcintosh@appcomsci.com

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 [25], 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 [18], 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 [1, 4, 9–11, 13–17, 19, 20, 26].

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 [10, 16, 17, 20, 26] can be hard to learn and use, or are simply limited in terms of expressiveness. For example, Secure Computation API (SCAPI) [4] does not provide a high-level language to specify the function to compute. Sharemind [2] fails to support conditional statements branching on private variables according to the user manual. Obliv-C [26], 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 [10]), 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 Sect. 5.1, some previous compilers [13, 14] require more than 2000 s 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 [7] 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 online¹.

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 [8, 16, 17]. In particular, the total work required

¹ <https://github.com/wangxiao1254/Secure-Computation-of-MIPS-Machine-Code>.

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 [8] (in an amortized sense); we show an example of this in Sect. 5.3.

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 $30\times$ on some programs, bringing our system to the point where it is feasible. In fact, for certain applications we are only $\sim 25\%$ slower than ObliVM [17] (see Sect. 5.3).

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. [21] 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. [5] 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 [12] 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 Sect. 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.

Our emulator uses as a building block Yao’s garbled-circuit protocol [25] 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 [7] 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. [8] 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 [8] can be used to prove that our construction is secure (in the semi-honest model) with respect to standard definitions of security [6], 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 Sect. 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 Sect. 3.1. In Sect. 3.2, we review relevant aspects of the MIPS architecture, and in Sect. 3.3, we give a high-level overview of how our system works. We provide some low-level details in Sects. 3.4 and 3.5. We defer until Sect. 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 Sect. 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 [17]. However, other frameworks [3, 21, 26] could also potentially be used (in conjunction with other ORAM implementations).

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

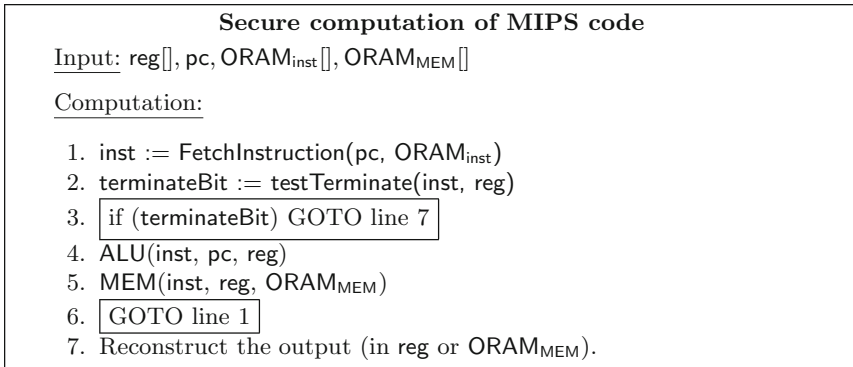


Fig. 1. Overview of secure computation of a MIPS program. Boxed lines are executed outside of secure computation.

3.5 Main Execution Loop

We use $\text{ORAM}_{\text{inst}}[], \text{ORAM}_{\text{MEM}}[], \text{reg}[], \text{pc}$, 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 Fig. 1, secure execution of a MIPS program involves repeated calls of three procedures: instruction fetch, ALU computation, and memory access.

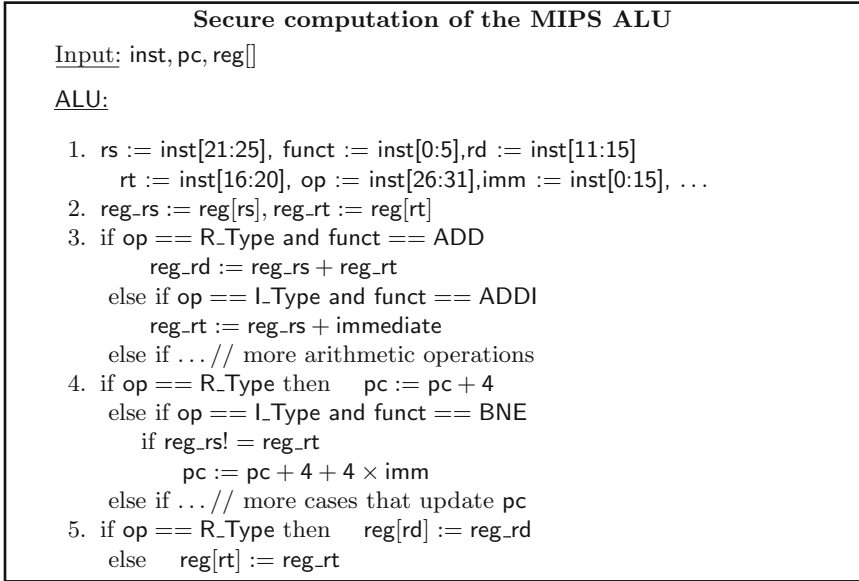


Fig. 2. This functionality takes the current instruction, program counter, and registers as input. Depending on the type of the instruction, it performs computation and updates the registers, as well as updating the program counter.

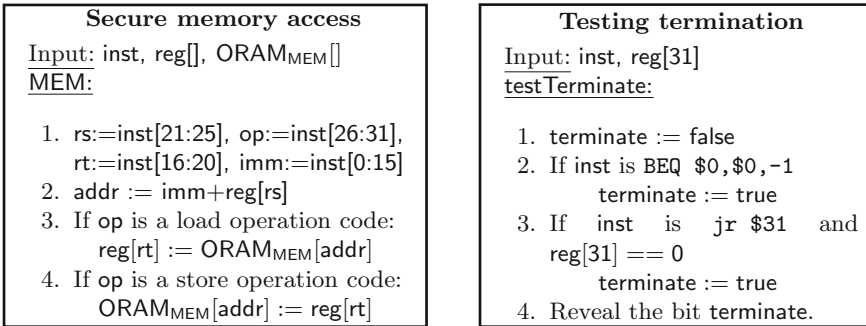


Fig. 3. (a) This functionality takes the instruction, the registers, and the memory as input and accesses the memory and registers according to the instruction. (b) This functionality takes the current instruction and the registers as input. It returns true if these indicate program termination.

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 Fig. 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 $\text{inst}[s:e]$ to denote the s -th bit to the e -th bits of inst .)
2. Retrieve values $\text{reg_rs} = \text{reg}[\text{rs}]$ and $\text{reg_rt} = \text{reg}[\text{rt}]$ 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.

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 Fig. 3a, 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 Fig. 3b.) 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 Sect. 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.

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: $1, 2, \dots, L, k + 1, \dots, L, k + 1, \dots, L, \dots$, until the termination condition is satisfied. In Sect. 5.1, we will discuss in detail how long such static analysis takes on programs of different sizes.

To illustrate this procedure, we provide a very simple example in Fig. 4a. 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 Fig. 4b, 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.

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 Sect. 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 Sect. 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

Example 1	Example 2
<pre> 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: </pre>	<pre> 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 </pre>

Fig. 4. (a) Assigning instructions to time steps. MULT instruction is not included in any ALU circuit but step 1. (b) An example demonstrating how we map instructions to time steps in a program with loops.

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 Sect. 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; 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 Fig. 4b. 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 $x + k_1, x + k_2, x + k_1 + k_2, \dots$, 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 Sect. 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

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 a GCC 4.7.3 compiler with `-O2 -std=c99 --save-temps` flags. When specified, we added NOPs to the binary by hand to explore the potential speedup introduced by padding; for all other examples, including binary search, decision trees, and Dijkstra’s shortest-path algorithm, we directly used the binaries generated by the GCC compiler without adding any padding. More detailed performance analysis and results for the last two examples can be found in the extended version [22].

All times reported are based on two machines of type `c4.4xlarge` with 3.3 GHz cores and 30 GB memory running in the same region of an Amazon EC2 cluster. Two machines are connected by ethernet with about 1.1 Gbps bandwidth. 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. Note that our system is based on OblivM, with a garbling rate of about 600K AND gates per second. A better garbled circuits implementation can therefore improve the absolute running time. Therefore we report different metrics for a better comparison.

Table 2. Total running time for computing the size of a set intersection.

Input size per party		Memory Access	Instruction Fetch	ALU Computation	Setup	Total Cost
64 Elements	Baseline	18.7 s	25.3 s	14.2 s	0.1 s	58.4 s
	+Inst. Mapping	5.4 s	3.6 s	3.8 s	0.1 s	12.9 s
	+Padding	0.5 s	0.4 s	1.8 s	0.1 s	2.8 s
256 Elements	Baseline	175.9 s	93.6 s	54.5 s	0.2 s	324.1 s
	+Inst. Mapping	51.9 s	14.4 s	14.5 s	0.2 s	81.0 s
	+Padding	4.9 s	1.3 s	6.6 s	0.2 s	13.0 s
1024 Elements	Baseline	2477.4 s	375.2 s	215.2 s	0.4 s	3068.2 s
	+Inst. Mapping	782.5 s	59.4 s	58.7 s	0.4 s	901.0 s
	+Padding	76.3 s	5.5 s	26.3 s	0.4 s	108.5 s

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. We ran each experiment 10 times. Since the standard derivation is never greater than 5% of the mean, we report the mean of 10 runs if not specified.

5.1 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 s 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 s 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. [13, 14] both take more than 2000 s 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 s, independent of the size of the matrices.

5.2 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:

- **Baseline.** This is the basic system (cf. Sect. 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 Sect. 4.1.
- **+Padding.** The program is padded (manually) with NOPs as described in Sect. 4.2.

For evaluation we use a program that computes the size of the intersection of two sets. Each party has a sorted array of 32-bit integers as input and wants to compute how many elements are shared by both parties. We report the total running time of the system in Table 2. For all running times shown, including the baseline, we incorporated the optimization that checks termination less frequently, as described in Sect. 4.3.

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

5.3 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 Gordon et al. [8] and Wang et al. [24], both of whom used hand-crafted circuits.

For testing our system, we wrote C code for binary search using a standard iterative implementation (see Fig. 5b), with no annotations or special syntax, and then compiled this to a MIPS binary using GCC. In Fig. 5a, we show the performance of our system when emulating the resulting MIPS binary, when using OblivM with Circuit ORAM [23] or trivial ORAM as the back-end. In addition, we implemented secure binary search using OblivM directly 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 Fig. 5a, our emulator with Circuit ORAM outperforms the one with trivial ORAM once the array contains $>2^{16}$ 32-bit integers, and outperforms a linear

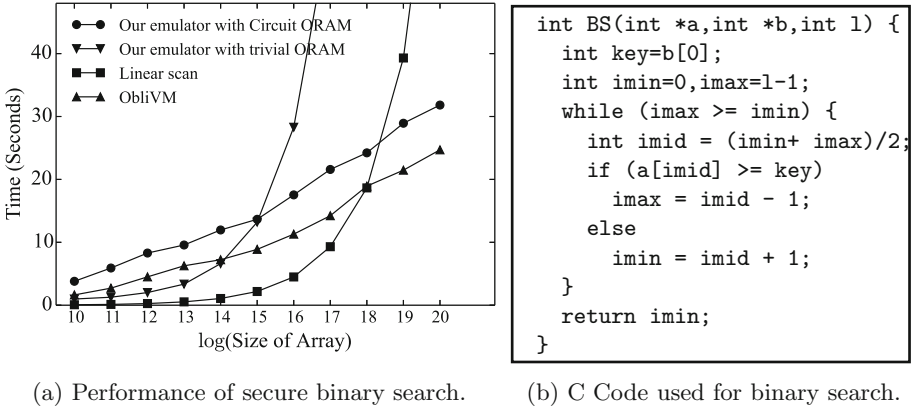


Fig. 5. Binary search example. One party holds an array of 32-bit integers, while the other holds a value to search for.

scan when the array contains $>2^{19}$ 32-bit integers. We can also see that our emulator is only 25% slower than OblivVM, which requires the programmer to use the domain-specific language supported by OblivVM.

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

1. Afshar, A., Mohassel, P., Pinkas, B., Riva, B.: Non-interactive secure computation based on cut-and-choose. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 387–404. Springer, Heidelberg (2014)
2. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: a framework for fast privacy-preserving computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008)
3. Demmler, D., Schneider, T., Zohner, M.: ABY—a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
4. Ejjenberg, Y., Farbstain, M., Levy, M., Lindell, Y.: SCAPI: the secure computation application programming interface. Cryptology ePrint Archive, Report 2012/629 (2012)
5. Fletcher, C.W., van Dijk, M., Devadas, S.: Towards an interpreter for efficient encrypted computation. In: Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop (2012)
6. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, Cambridge (2004)
7. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. *J. ACM* **43**(3), 431–473 (1996)
8. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS, pp. 513–524. ACM Press, October 2012
9. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM CCS, pp. 451–462. ACM Press, October 2010
10. Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS, pp. 772–783. ACM Press, October 2012
11. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: Usenix Security Symposium (2011)
12. Keller, M.: The oblivious machine - or: how to put the c into mpc. Cryptology ePrint Archive, Report 2015/467 (2015). <http://eprint.iacr.org/>
13. Kreuter, B., Mood, B., Shelat, A., Butler, K.: PCF: a portable circuit format for scalable two-party secure computation. In: Usenix Security Symposium (2013)
14. Kreuter, B., Shelat, A., Shen, C.H.: Billion-gate secure computation with malicious adversaries. In: USENIX Security Symposium (2012)
15. Lindell, Y., Riva, B.: Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In: ACM CCS 2015, pp. 579–590. ACM Press (2015)
16. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.: Automating efficient RAM-model secure computation. In: IEEE Security & Privacy (2014)
17. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: Oblivm: a programming framework for secure computation. In: IEEE Security & Privacy (2015)

18. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay: a secure two-party computation system. In: USENIX Security Symposium (2004)
19. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009)
20. Rastogi, A., Hammer, M.A., Hicks, M.: Wysteria: a programming language for generic, mixed-mode multiparty computations. In: 2014 IEEE Symposium on Security and Privacy, pp. 655–670. IEEE Computer Society Press, May 2014
21. Songhori, E.M., Hussain, S.U., Sadeghi, A.R., Schneider, T., Koushanfar, F.: TinyGarble: highly compressed and scalable sequential garbled circuits. In: IEEE Security & Privacy (2015)
22. Wang, X., Gordon, S.D., McIntosh, A., Katz, J.: Secure computation of mips machine code. Cryptology ePrint Archive, Report 2015/547 (2015). <http://eprint.iacr.org/2015/547>
23. Wang, X.S., Chan, T.H., Shi, E.: Circuit oram: on tightness of the goldreich-ostrovsky lower bound. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM (2015)
24. Wang, X.S., Huang, Y., Chan, T.H.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS, pp. 191–202. ACM Press, November 2014
25. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS, pp. 162–167. IEEE Computer Society Press, October 1986
26. Zahur, S., Evans, D.: Obliv-c: a language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153 (2015)