GarbledCPU: A MIPS Processor for Secure Computation in Hardware

Ebrahim M. Songhori
Rice University
Houston, TX, USA
ebrahim@rice.edu

Shaza Zeitouni
Technische Universität
Darmstadt, Germany
shaza.zeitouni@trust.tu-darmstadt.de

Ghada Dessouky
Technische Universität
Darmstadt, Germany
ghada.dessouky@trust.tu-darmstadt.de

Thomas Schneider
Technische Universität
Darmstadt, Germany
thomas.schneider@crisp-da.de

Ahmad-Reza Sadeghi
Technische Universität
Darmstadt, Germany
ahmad.sadeghi@cased.de

Farinaz Koushanfar
University of California
San Diego, CA, USA
farinaz@ucsd.edu

ABSTRACT

We present GarbledCPU, the first framework that realizes a hardware-based general purpose sequential processor for secure computation. Our MIPS-based implementation enables development of applications (functions) in a high-level language while performing secure function evaluation (SFE) using Yao's garbled circuit protocol in hardware. GarbledCPU provides three degrees of freedom for SFE which allow leveraging the trade-off between privacy and performance: public functions, private functions, and semi-private functions. We synthesize GarbledCPU on a Virtex-7 FPGA as a proof-of-concept implementation and evaluate it on various benchmarks including Hamming distance, private set intersection and AES. Our results indicate that our pipelined hardware framework outperforms the fastest available software implementation.

Keywords

Garbled Circuit; Secure Function Evaluation

1. INTRODUCTION

Secure Function Evaluation (SFE) allows two (or more) mistrusting parties to jointly compute an arbitrary function on their private inputs without revealing information but the result. The seminal work of Yao [20] has introduced the concept of two-party SFE using the Garbled Circuits (GC) protocol which requires that the function is represented as a Boolean circuit. While the GC protocol was originally thought to be of theoretic interest only, algorithmic and implementation optimizations have significantly improved its efficiency during the last decades. In addition to the advances in computing platforms, the key enablers for the progress include newer cryptographic constructs, logic-level transformations, and software techniques.

Compilers for SFE have been continually evolving. A number of compilers [2, 4, 12, 13] translate a functionality written in a domain-specific input language into a Boolean circuit, also described in an intermediate language, which is then evaluated with Yao's GC protocol. Other compilers [5, 11] use a subset of the C language as input. However, these methods imply building software-to-Boolean circuit compilers from scratch and often put limitations on the functionality. Moreover, verifying the correctness of these compilers is challenging [14].

Recently, it was shown that the long-established and verified hardware synthesis compilers can be used for generation of Boolean circuits for SFE, eliminating the need for building ad-hoc logic compilers or tedious handcrafting of Boolean circuits. Another key advantage of conventional logic synthesis is allowing a *sequential logic description* which can be adapted to map general functionalities to Boolean circuits optimized for Yao's GC protocol. The approach was introduced in [17] and shown to yield great improvements in terms of memory and communication. A fully-automated toolchain which utilizes existing logic synthesis compilers and can be generalized for other SFE protocols was presented in [3]. This latter work takes advantage of the built-in intellectual property (IP) and custom design libraries which can be readily adapted during circuit synthesis to realize a broad suite of applications optimized for SFE.

The authors in [17] leverage the capability of synthesizing a sequential circuit and introduce the idea of a general-purpose sequential processor for private function SFE (PF-SFE) by GC, where both input data and function are private. PF-SFE is useful for scenarios where the function is proprietary or classified, e.g., credit checking or private database queries. Their so-called *garbled processor* allows to use existing software compilers for describing the function and generates compatible machine code which is also garbled. A partial implementation of a MIPS processor is provided in [17] which only considers PF-SFE where the entire processor circuit and Instruction Set (IS) have to be garbled in each instruction step during SFE in order to hide the executed instructions in the private function. This re-

DAC '16, June 05-09, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897937.2898027

sults in a tremendous cost compared with SFE for public functions. Thus, their MIPS processor incurs an unnecessary overhead for numerous applications in which a private function is not required. (The only benchmark presented in [17] is the Hamming distance function.)

We propose GarbledCPU, the first configurable hardware-based general purpose sequential CPU for SFE. The FPGA realization of GarbledCPU is based on the MIPS instruction set. GarbledCPU provides a generalized support for SFE of varying flavors of privacy, beyond PF-SFE, to allow for more relaxed privacy demands and hence an improved performance. More explicitly, with GarbledCPU the parties can evaluate a private, semi-private or public function by revealing none, partial or all information about the function respectively while still benefiting from the simplicity of programming a processor. Both parties decide first which subset of IS they are willing to use which determines the level of privacy ensured. The function is compiled from a high-level language, e.g., C/C++ into assembly code of the agreed upon IS. Next, the garbled processor is securely evaluated given users' garbled input and the compiled function instructions (also garbled) to compute the output.

A recent technical report [19] also suggests a secure computation framework using MIPS code. The approach relies on garbled universal circuits to emulate the execution of each instruction of the MIPS program and on Oblivious RAM (ORAM) for memory access. They propose using static analysis of functions to reduce the set of instructions to be garbled. However, [19] only presents a software SFE implementation, while we present the first practical hardware sequential processor for both SFE and PF-SFE. An earlier hardware implementation of GC was reported in [7] but the approach only addresses SFE with no support for function hiding and is limited to combinational Boolean circuit as it predated [17]. A combinational description limits usability and scalability and is impractical for control-intensive functions such as CPUs that need to be expressed sequentially.

Contributions. In brief, our contributions are as follows:

- We propose the first hardware-only solution for 2-party GC-based secure sequential function computation with different SFE flavors that allows leveraging the trade-off between privacy and performance: application-specific IS for SFE (§3.1), restricted IS (§3.2) for semi-private SFE, and full IS (§3.3) for PF-SFE.
- We realize a proof-of-concept FPGA implementation which demonstrates the feasibility of the sequential garbled processor in hardware, and motivates further research in this direction. GarbledCPU achieves efficiency and performance by leveraging the most recent optimizations for GC [1, 9, 17, 21], along with a high-throughput pipelined GC evaluation on FPGA. It outperforms the fastest software implementation in the literature which relies on the Intel AES-NI [1].
- We extensively benchmark more complex functions such as AES, Private Set Intersection (PSI), and Hamming distance and evaluate them under our different privacy settings using our framework and when applicable, compare our performance with prior work.

2. PRELIMINARIES

In this section, we present an introduction to secure computation and SFE in §2.1 and GC optimizations in §2.2.

2.1 Secure Computation and Garbled Circuit

Yao's GC protocol [20] allows two parties, *Alice* and *Bob*, to jointly compute a function $f(x_{Alice}, x_{Bob})$ on their private inputs (x_{Alice} and x_{Bob}). Alice *garbles* the function f, where f is represented as a Boolean circuit. To do this, Alice maps the plain binary values of inputs and intermediate gates' outputs to random *labels* (keys). For each gate in the circuit, an encrypted truth table is generated that allows computation of the gate's output label based on its input labels. Alice sends the encrypted truth tables of all gates, along with her corresponding encrypted input labels to Bob. To compute f, Bob needs to know the labels corresponding to his inputs without revealing them to Alice. For this, Bob obtains his labels obliviously through a 1-out-of-2 Oblivious Transfer (OT) protocol [15] and uses them to *evaluate* the garbled circuit gate by gate. Finally, Alice provides a mapping from the encrypted output label to the plain output.

Two-party Private Function SFE (PF-SFE) allows secure computation of a function $f_{Alice}(\cdot)$ held by Alice on Bob's data x_{Bob} (Bob) while both the data and the function are kept private, i.e., Bob learns $f_{Alice}(x_{Bob})$ but nothing else about f_{Alice}. This is in contrast to the usual setting of SFE where the function is known to both parties. PF-SFE is especially useful when the function is proprietary or classified. Traditionally, PF-SFE is realized by running SFE of a Universal Circuit (UC) [8, 10, 18]. A UC is similar to a Universal Turing Machine that receives a Turing machine description $f(\cdot)$ and applies it to the input data on its tape. Usage of UC results in complexity at least $\mathcal{O}(n \log n)$ for a circuit with n gates [18].

2.2 GC Optimizations

Our GC evaluator architecture is based on fixed-key block cipher garbling [1] and utilizes garbling sequential circuits [17]. We also consider the most recent optimizations on GC: the half-gates technique [21] allows to use two ciphertexts for each non-XOR gate (instead of three) while still being compatible with the free-XOR technique [9].

3. GARBLED PROCESSOR

The idea of garbling a processor was first introduced in [17] as a solution for hiding the function in PF-SFE. Besides enabling PF-SFE, another advantage of a garbled processor is usability for non-expert users since it can be programmed using high-level languages, whereas other frameworks for the GC protocol require tedious Boolean circuit construction. However, garbling and evaluating the entire processor incurs a tremendous cost compared to SFE solutions due to stronger privacy requirements in PF-SFE.

Adversary Model. We assume an honest-but-curious (i.e., semi-honest or passive) adversary which is sufficient for most practical scenarios to enable efficient protocols. This establishes the first step towards protocols with stronger security guarantees against malicious or covert adversaries.

In this work, we propose GarbledCPU as a hardware-only garbled processor framework for secure computation that provides scalable support for generalized SFE with a relaxed privacy setting but improved performance, besides the more security-demanding PF-SFE, as well as a flavor in-between. To avoid information leakage about the function (i.e., PF-SFE), we use GarbledCPU with its full Instruction Set (IS),

which incurs a large overhead due to garbling and evaluating of the entire IS. We can also compile the function using only a subset of the IS: restricted IS (i.e., semi-private function). A third alternative is public function mode in which the function is compiled using only an application-specific subset of the IS that is required for executing the function. In the following, we discuss these modes of function evaluation and the trade-off between privacy and performance further.

Figure 1 shows the overview of GarbledCPU for 2-party computation between Alice (garbler) and Bob (evaluator). Alice generates the garbled instructions and tables by garbling the processor circuit for the selected IS mode and sends them to Bob. He also receives his garbled input data through OT from Alice without revealing his input to her. Bob evaluates GarbledCPU and produces the garbled output. Eventually, Alice reveals the output map to Bob and he "ungarbles" and learns the output data.

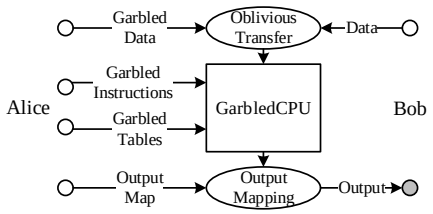

Figure 1: Overview of GarbledCPU.

3.1 Garbled Processor for Public Functions

Using a general-purpose processor with its entire IS in SFE results in garbling a large processor which is very costly and unnecessary since both parties know the function instructions being executed but not their results. Hence, garbling a limited application-specific IS for executing each instruction is sufficient to achieve privacy. In §5.3 we show three examples of GarbledCPU with application-specific IS. To further reduce the IS, assuming for example, a function that consists of 10 instructions, we could theoretically generate $2^{10} - 1$ netlists (netlists of IS with different combinations of the 10 instructions, excluding the netlist with zero instructions). At run-time, one of these netlists is plugged in (garbled and evaluated) at each instruction step depending on the expected instructions. However, to make it more reasonable (generate fewer netlists), for functions with control flow independent of private data, we know in advance which instruction will be executed at each step. Thus, we need only the netlist of the processor implementing IS with that specific instruction, restricting the required netlists in this case to 10. For functions with control flow dependent on private data, a simple static analysis can be used to specify the combination of possible instructions at each step, and hence the required IS netlist as proposed in [19].

3.2 Garbled Processor for Semi-Private Functions

The main cost for garbling a processor with its entire IS results from garbling circuits for expensive instructions like multiplication and division. Most compilers are able to avoid these costly instructions and replace them with cheaper loops of shifts, addition, and subtraction instructions. This would eliminate the need for the Mult/Div unit in the processor and reduce the cost of garbling per instruction on one hand. However on the other hand, one expensive instruction will be replaced with multiple cheap instructions, thus increasing the total number of instructions. For example, multiplying two 32-bit numbers with the MULT instruction in MIPS requires 15 cycles and a circuit of 13,257 non-XOR gates[1], while it requires at least 31 cycles and a circuit of 9.676 non-XOR gates when using a conditional loop over an ADD instruction. We call this mode "semi-private" since it only reveals partial information about instructions used in the program (that the program does not use division/-multiplication) and increases the probability of guessing an instruction by reducing the subset of possible instructions (restricted IS).

3.3 Garbled Processor for Private Functions

In the standard 2-party PF-SFE, Alice provides the function $f_{Alice}(\cdot)$ and Bob provides the input data x_{Bob} and the output is $f_{Alice}(x_{Bob})$. In this work, GarbledCPU receives a list of *instructions* as $f_{Alice}(\cdot)$ and applies them to the input data x_{Bob} in memory and the output will be written back to the memory. To avoid information leakage about the private function, we use a general-purpose processor with its entire IS (full IS).

4. GarbledCPU IMPLEMENTATION

We present our garbled processor in §4.1, a high performance hardware architecture for GC to evaluate GarbledCPU in §4.2, and implementation challenges in §4.3.

4.1 MIPS

To implement GarbledCPU, we use the MIPS architecture from the Plasma project in Opencores [16]. We chose the single-cycle implementation of the 32-bit MIPS I instruction set which is based on the Reduced Instruction Set Computing (RISC), making its Boolean representation among the simplest of modern processors. Note that the gates should be garbled/evaluated one after another in the GC protocol, and it is challenging to benefit from physical level parallelism that exists inherently in hardware. Thus, using a multi-cycle, pipelined, or a more sophisticated architecture not only complicates the implementation, but also counter-intuitively, increases the overall cost of garbling for the same functionality. The time required for garbling a circuit depends only on the total number of gates and not the critical path. In §5.3 we present the garbling cost for MIPS with various memory sizes.

4.2 Our Hardware Architecture

To the best of our knowledge, the fastest implementation of GC is JustGarble [1]. JustGarble is a software GC realization using fixed-key AES garbling which benefits from the AES-NI instruction set in modern Intel processors. Its performance reaches about 20 clock cycles per gate for GC evaluation. Prior to JustGarble, Järvinen et al. introduced two hardware realizations for the GC protocol in [7]. However, their performance is much slower than JustGarble because JustGarble utilizes a more efficient fixed-key AES for garbling instead of an expensive hash function. Thus, it is possible that a hardware implementation leveraging the latest GC optimizations including fixed-key AES garbling would outperform JustGarble. Furthermore, a processor is

[1]XOR gates are evaluated freely in GC according to the free-XOR optimization of [9].

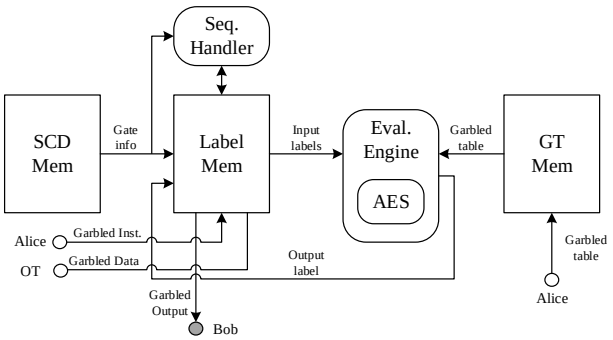

Figure 2: Our GC Evaluator Architecture.

essentially a sequential circuit and its evaluation requires sequential GC which none of these works supports.

Our GC evaluator is based on the most recent optimizations listed in §2.2. Its architecture is shown in Figure 2 and consists of: (1) Simple Circuit Description (SCD) memory: read-only memory that stores the information about gates in the MIPS circuit in SCD format [1, 17]. (2) GC Label memory: read-write random-access memory that stores GC ciphertext labels of all wires in the corresponding MIPS circuit. (3) Garbled Tables (GT) memory: read-write random-access memory that stores the ciphertext garbled tables of each non-XOR gate in the MIPS circuit that are generated by Alice (garbler). (4) Sequential Handler: controller that supports evaluation of the sequential circuits with the GC protocol. (5) Evaluator Engine: main functionality of GC evaluation according to Yao's GC protocol and its most recent optimizations [1, 9, 17, 21].

As shown in Figure 2, Bob's input labels in the Label memory are initialized by the OT protocol with Alice. The rest of the labels in the Labels memory and the Garbled Tables memory are received in clear-text from Alice.

Pipelined Evaluator Engine and Gate Dependency. To maximize the performance of the GC evaluator, we use a 20-stage pipelined AES implementation [6] inside our Evaluator Engine module. It increases the throughput of the module by increasing the maximum operating clock frequency of the engine. We also add one stage for the rest of the GC evaluation functionality. Due to the free-XOR technique [9], evaluating an XOR gate requires only XOR-ing the input labels while evaluating a non-XOR gate requires two AES encryptions (due to half-gates technique presented in §2.2, and was one encryption before). Therefore, evaluation of an XOR gate can be done in only one stage of the AES pipeline. Different timing for XOR and non-XOR gates introduces a challenge for handling dependencies of gates' inputs and output. A gate cannot enter the evaluation pipeline if its inputs are another gate's output which is not yet evaluated. This results in pipeline stalls which degrade the overall performance. To mitigate this, we push XOR gates to the latest empty stage of the pipeline such that the subsequent dependent gates can enter the pipeline as soon as possible.

4.3 Hardware Prototype Challenges

We only use on-chip memory for our proof-of-concept in this work. However, this prototype can be extended to support interfacing with off-chip memory which would store garbled tables and labels of larger garbled processor circuits and functions. It can also interface with another FPGA emulator of the garbler which generates the garbled tables and labels and streams them to our evaluator. A wide range of scenarios are now feasible owing to our current hardware platform and state-of-the-art optimized GC evaluator. Such extensions would incur additional area and performance overheads, but would allow upscaling of our implementation to support garbled processor circuits and benchmarks in the Gigabytes range. We emphasize that we provide in this work a proof-of-concept prototype to motivate further research in this direction to bring garbled processors some steps closer to the realm of feasible practical implementations.

5. EVALUATION

We give our evaluation setup in §5.1, benchmarks in §5.2, synthesis results in §5.3, and performance evaluation in §5.4.

5.1 Evaluation Setup

We create different instances of a single-cycle MIPS architecture with specific, restricted, and full IS to support a trade-off between efficiency and privacy. (Full IS was proposed in [17] and is reported for comparison.) The different MIPS instances are synthesized using Synopsys Design Compiler DC H-2013.03-SP4 to generate optimized sequential Boolean circuits. These circuits are then evaluated on our hardware GC evaluator implemented using Vivado 2014.4.1 on a Xilinx Virtex-7 FPGA.

5.2 Benchmarks

As benchmarks we used Hamming distance, private set intersection and AES. We compile these benchmarks from high-level C to MIPS binary using a MIPS cross-compiler. For some benchmarks, assembly code manipulation allows to reduce the number of clock cycles required. To assure correctness of both benchmarks and IS under test, we simulate the resulting binary file using the Modelsim simulator and calculate the number of required cycles to compute each of the benchmarks, reported in Table 1, for accurate performance measurements. For Hamming distance, the number of cycles depends on the size of the input strings. In the PSI benchmark, we compute a variant of PSI called PSI cardinality (PSI-CA) where only the number of common elements is revealed. The sets can have different sizes where each element is 32-bit. For AES, we assume that one party holds a 128-bit message and the other party holds eleven round keys each of 128-bit length to avoid unnecessary garbling and evaluation of the round-key generation function.

Table 1: Number of required cycles to compute benchmarks.

Benchmark	Input Size	# of required cycles
Hamming Distance	16	218
Hamming Distance	32	426
Hamming Distance	64	842
Hamming Distance	128	1,674
PSI	64	7,267
PSI	128	14,267
AES (no key expansion)	128	6,178

5.3 Synthesis of the GarbledCPU IS

We synthesize the MIPS architecture, shown in Figure 3, with Synopsys DC for different ISs and memory sizes: 32 to 512 32-bit words for instruction and data memories. Generating these Boolean circuits is a one-time process and the circuits can be re-used without incurring further compila-

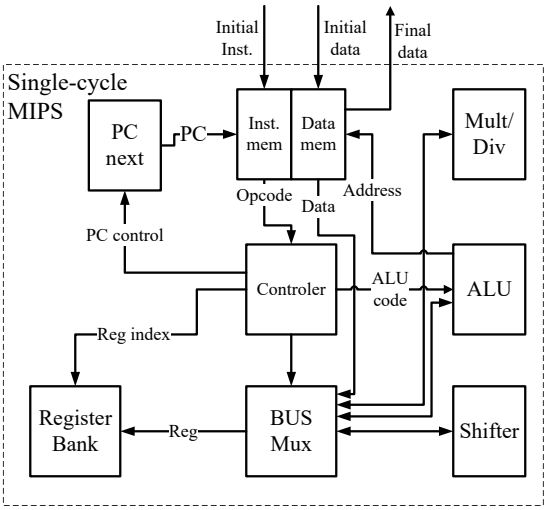

Figure 3: Single-Cycle MIPS architecture.

tion costs. Table 2 shows the synthesis time and number of non-XOR gates of the IS's with different sizes of memories.

Table 2: Synthesis results of different variants of (IS)

Memory Size (words)	Synthesis Time seconds (s)	# of Combinatorial non-XOR gates	# of Sequential gates
Hamming Distance-specific IS			
DM, IM = 32	19.648 *s*	6,715	2,021
DM, IM = 64	31.692 *s*	9,830	3,046
DM, IM = 128	62.212 *s*	16,062	5,095
DM, IM = 256	167.398 *s*	28,493	9,192
DM, IM = 512	589.186 *s*	53,374	17,385
PSI-specific IS			
DM, IM = 32	18.735 *s*	6,751	2,021
DM, IM = 64	31.117 *s*	9,866	3,046
DM, IM = 128	61.551 *s*	16,097	5,095
DM, IM = 256	163.564 *s*	28,529	9,192
DM, IM = 512	591.145 *s*	53,410	17,385
AES-specific IS			
DM, IM = 256	169.498 *s*	32,177	9,214
DM, IM = 512	594.047 *s*	61,570	17,406
ALU&Shift IS			
DM, IM = 32	21.681 *s*	9,676	2,046
DM, IM = 64	34.588 *s*	12,702	3,070
DM, IM = 128	65.873 *s*	19,694	5,118
DM, IM = 256	170.974 *s*	34,071	9,214
DM, IM = 512	593.945 *s*	66,238	17,406
ALU-only IS			
DM, IM = 32	19.599 *s*	8,136	2,046
DM, IM = 64	31.970 *s*	11,696	3,070
DM, IM = 128	62.599 *s*	18,816	5,118
DM, IM = 256	164.894 *s*	33,041	9,214
DM, IM = 512	598.986 *s*	65,183	17,406
Full IS [17]			
DM, IM = 32	38.331 *s*	13,257	2,110
DM, IM = 64	50.446 *s*	16,818	3,134
DM, IM = 128	82.863 *s*	23,899	5,182
DM, IM = 256	189.157 *s*	38,118	9,278
DM, IM = 512	616.750 *s*	69,423	17,470

- *Application-specific IS for public functions:* We synthesized three variants of the application-specific IS where the selected instructions include only the ones used by a particular function, for various memory sizes. We create the application-specific IS for the three benchmarks: Hamming distance[2], PSI[3], and AES[4].

[2]Instructions required for *Hamming distance* are: LW, SW, ADD, SUB, XOR, NOP, SLL and BEQ

[3]Instructions required for *PSI* are: LW, SW, ADD, SUB, NOP, SLL, BEQ, BNE and SLT

[4] Instructions required for *AES* are: LW, LB, SW, SB, ADD, SUB, AND, XOR, OR, NOP, SLL, SRL, BEQ, BNE, JAL, JR and SLT

- *Restricted IS for semi-private functions:* We synthesized two variants of the restricted IS: one without the Mult/Div unit and another without Mult/Div and Shift units. Since the difference between the two depends mainly on reducing the control logic and select lines of multiplexers, the numbers of non-XOR gates for both are different. However, the number of flip-flops are the same.
- *Full IS [17] for private functions:* We show full IS synthesis results with different memory sizes in Table 2.

5.4 Performance Evaluation

Area. Table 3 shows the resource allocation and utilization of our GC Evaluator on a Xilinx Virtex-7 FPGA. Note that the FPGA utilization does not vary for different memory sizes and instances of the MIPS processor since the evaluator logic remains unaltered. For different memory sizes and IS instances, only the non-XOR gate count varies. This only impacts the garbled labels and tables memory which significantly affects the off-chip memory utilized for storing the garbled tables, and the Block Random-Access Memory (BRAM) resources utilization only to a small extent.

Table 3: Resource allocation and utilization of GarbledCPU GC Evaluator on a Xilinx Virtex-7 FPGA.

Resource	Estimation	Utilization %
Flip-Flop (FF)	22,035	2.54
Slice LookUp Table (LUT)	21,229	4.90
BRAM	354	24
BUFG	2	6.25

Performance. Table 4 presents the runtime required to evaluate GarbledCPU for one instruction in terms of clock cycles and μs. Our GC evaluator operates at 100MHz on the FPGA. This is used to compute an average evaluation runtime of 1.1 clock cycles per gate for our pipelined GC evaluator which translates to an average of 11ns per gate in our FPGA implementation. The reported runtime can be further improved by providing tighter timing constraints.

Comparison with Other Work. Table 5 shows a comparison with other GC evaluator implementations. However, for fairness, we are leveraging GC optimizations that were not available at the time for [7]. We compare with our two implementations, the 21-stage pipelined evaluator and unpipelined variant to show the effect of pipelining in improving our performance by a factor of 7.8. Table 5 compares our results with interpolated results estimated for other works. Results indicate that our pipelined GC evaluator FPGA implementation takes 51× fewer clock cycles compared to the fastest software implementation JustGarble [1]. Although the CPU clock frequency (3.0GHz) is 30× faster than that of our Virtex-7 FPGA (100MHz), our pipelined implementation would still be almost 2× faster than JustGarble in terms of absolute time. Note that our implementation is just a prototype on a reconfigurable FPGA as opposed to a custom design of Intel AES-NI in CPU. Implementing GarbledCPU on an ASIC would improve its performance in terms of absolute time even further. Moreover, our implementation is two orders of magnitude faster than the previously fastest hardware implementation of [7].

Table 4: Performance of GarbledCPU for different (ISs) with different memory sizes at 100MHz clock frequency.

Memory Size (words)	32	64	128	256	512
Hamming Distance-IS					
# of non-XOR gates	6,715	9,830	16,062	28,493	53,374
Time per inst. (cc)	7,118	10,813	17,829	30,773	57,644
Time per inst. (μs)	71.18	108.13	178.29	307.72	576.44
Avg. Time per gate (cc)	1.06	1.10	1.11	1.08	1.08
PSI-IS					
# of non-XOR gates	6,751	9,866	16,097	28,529	53,410
Time per inst. (cc)	7,426	10,952	18,029	30,811	57,149
Time per inst. (μs)	74.26	109.52	180.29	308.11	571.49
Avg. Time per gate (cc)	1.10	1.11	1.12	1.08	1.07
AES-specific IS					
# of non-XOR gates	-	-	-	32,177	61,570
Time per inst. (cc)	-	-	-	35,717	68,343
Time per inst. (μs)	-	-	-	357.17	683.43
Avg. Time per gate (cc)	-	-	-	1.11	1.11
ALU&Shift-IS					
# of non-XOR gates	9,676	12,702	19,694	34,071	66,238
Time per inst. (cc)	10,644	13,972	22,057	36,115	71,537
Time per inst. (μs)	106.44	139.72	220.57	361.15	715.37
Avg. Time per gate (cc)	1.10	1.10	1.12	1.06	1.08
ALU-only IS					
# of non-XOR gates	8,136	11,696	18,816	33,041	65,183
Time per inst. (cc)	8,624	12,866	21,074	35,684	73,657
Time per inst. (μs)	86.24	128.66	210.74	356.84	736.57
Avg. Time per gate (cc)	1.06	1.10	1.12	1.08	1.13
Full IS [17]					
# of non-XOR gates	13,257	16,818	23,899	38,118	69,423
Time per inst. (cc)	14,848	18,668	25,811	40,786	77,060
Time per inst. (μs)	148.48	186.68	258.11	407.86	770.60
Avg. Time per gate (cc)	1.12	1.11	1.08	1.07	1.11

Table 5: Comparing our GC evaluator implementation with other works' estimation for MIPS with 64-word memory.

Method	Total time (cc)	cc/gate
Järvinen et al. (SoC) [7]	37,329,233	2,219.6
Järvinen et al. (Stand-Alone FPGA) [7]	4,291,954	255.2
JustGarble (CPU) [1]	948,535	56.4
Our work w/o pipeline	144,635	8.6
Our work w/ pipeline	18,500	1.1

6. CONCLUSION

We introduce GarbledCPU, the first hardware realization of a sequential CPU for secure evaluation of MIPS code. GarbledCPU enables to evaluate either public, semi-private or private function on secret inputs by trading-off between privacy and performance. GarbledCPU is synthesized on a Virtex-7 FPGA as a proof-of-concept implementation, and we evaluated our framework for three benchmarks: Hamming distance, private set intersection, and AES.

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments. This work is partially supported by an Office of Naval Research grant (ONR-R17460), a National Science Foundation grant (CNS-1059416), and a Multidisciplinary University Research Initiative grant (FA9550-14-1-0351/ Rice 14-0538) to the ACES lab at Rice University. The work of the authors at TU Darmstadt is supported by the European Union's Seventh Framework Program (FP7/2007-2013) grant agreement n. 609611 (PRACTICE), the German Science Foundation (DFG) as part of project E3 within the CRC 1119 CROSSING, the German Federal Ministry of Education and Research (BMBF) within CRISP, and the Hessian LOEWE excellence initiative within CASED.

7. REFERENCES

[1] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P'13*. IEEE, 2013.

[2] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM CCS'08*. ACM, 2008.

[3] D. Demmler, G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, and S. Zeitouni. Automated synthesis of optimized circuits for secure computation. In *ACM CCS'15*. ACM, 2015.

[4] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM CCS'10*. ACM, 2010.

[5] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *ACM CCS'12*. ACM, 2012.

[6] H. Hsing. Tiny AES, 2013. http://opencores.org/project,tiny_aes.

[7] K. Järvinen, V. Kolesnikov, A. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *CHES'10*. Springer, 2010.

[8] A. Kiss and T. Schneider. Valiant's universal circuit is practical. In *EUROCRYPT'16*. Springer, 2016.

[9] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP'08*. Springer, 2008.

[10] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *FC'08*. Springer, 2008.

[11] B. Kreuter, A. Shelat, B. Mood, and K. R. B. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security'13*. USENIX, 2013.

[12] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security'12*. USENIX, 2012.

[13] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security'04*. USENIX, 2004.

[14] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *IEEE Euro S&P'16*. IEEE, 2016.

[15] M. O. Rabin. How to exchange secrets with oblivious transfer. TR-81, Aiken Computation Lab, Harvard University, 1981.

[16] S. Rhoads. Plasma-most MIPS I, 2006. http://opencores.org/project,plasma.

[17] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE S&P'15*. IEEE, 2015.

[18] L. G. Valiant. Universal circuits (preliminary report). In *STOC'76*. ACM, 1976.

[19] X. S. Wang, S. D. Gordon, A. McIntosh, and J. Katz. Secure computation of MIPS machine code. Cryptology ePrint Archive, Report 2015/547, 2015. http://eprint.iacr.org/2015/547.

[20] A. Yao. How to generate and exchange secrets. In *FOCS'86*. IEEE, 1986.

[21] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole. In *EUROCRYPT'15*. Springer, 2015.