2015 IEEE Symposium on Security and Privacy

TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits

Ebrahim M. Songhori*, Siam U. Hussain*, Ahmad-Reza Sadeghi†, Thomas Schneider†, Farinaz Koushanfar*
*Rice University, Houston, TX, USA
†Technische Universität Darmstadt, Darmstadt, Germany
{ebrahim, siam.umar}@rice.edu, ahmad.sadeghi@trust.cased.de, thomas.schneider@ec-spride.de, farinaz@rice.edu

***Abstract*—We introduce TinyGarble, a novel automated methodology based on powerful logic synthesis techniques for generating and optimizing compressed Boolean circuits used in secure computation, such as Yao's Garbled Circuit (GC) protocol. TinyGarble achieves an unprecedented level of compactness and scalability by using a sequential circuit description for GC. We introduce new libraries and transformations, such that our sequential circuits can be optimized and securely evaluated by interfacing with available garbling frameworks. The circuit compactness makes the memory footprint of the garbling operation fit in the processor cache, resulting in fewer cache misses and thereby less CPU cycles. Our proof-of-concept implementation of benchmark functions using TinyGarble demonstrates a high degree of compactness and scalability. We improve the results of existing automated tools for GC generation by orders of magnitude; for example, TinyGarble can compress the memory footprint required for 1024-bit multiplication by a factor of 4,172, while decreasing the number of non-XOR gates by 67%. Moreover, with TinyGarble we are able to implement functions that have never been reported before, such as SHA-3. Finally, our sequential description enables us to design and realize a garbled processor, using the MIPS I instruction set, for private function evaluation. To the best of our knowledge, this is the first scalable emulation of a general purpose processor.**

***Index Terms*—Secure Function Evaluation, Garbled Circuit, Logic Design, Hardware Synthesis**

I. Introduction

Secure function evaluation (SFE) allows two or more parties to correctly compute a function of their respective private inputs without exposure. The seminal result by Yao introduced the GC protocol for addressing two-party SFE [70]. The GC protocol allows to securely evaluate a function given as a Boolean circuit that is represented as a series of binary gates. The inputs and outputs of each gate are masked such that the party evaluating the GC cannot gain any information about the inputs or intermediate results that appear during function evaluation. The approach of obliviously evaluating a Boolean circuit can also be generalized to multi-party SFE [4], [26].

Contemporary literature has cited multiple important privacy preserving and security critical applications that could benefit from a practical realization of SFE, including but not limited to: biometrics matching, face recognition, image/data classification, electronic auctions and voting, remote diagnosis, and secure search [1], [8], [9], [22], [39], [58]. While GC was considered to be prohibitively expensive and practically infeasible a decade ago, today we are witnessing a surge of theoretical, algorithmic, and tool developments that have significantly improved the efficiency and practicality of the GC protocol, see [2], [36], [44], [54], [60].

The research on producing Boolean functions for SFE can be roughly classified into two categories: optimizations of cryptographic constructs and protocols such as [2], [3], [42], [44], [60], [72], and compiler/engineering techniques including but not limited to [23], [32], [36], [46], [47], [53], [54].

In the compiler/engineering realm two different approaches for circuit generation have been developed. One approach is based on building a custom library for a general purpose programming language such as Java along with functions for emitting the circuit, e.g., [32], [36], [53]. For better usability, these libraries typically include frequently used modules such as adders and multipliers. However, library-based approaches require manual adjustment and do not perform global circuit optimization. Moreover, their memory management gets complicated when the number of gates is large thereby affecting performance and scalability [32].

The second approach is to write a new compiler for a higher-level language that translates the instructions into the Boolean logic, e.g., [23], [46], [47], [54]. Although compiler-based approaches can perform global optimizations, they often unroll the circuits into a large list of gates. For example, the description of a circuit with one billion gates has at least size $2\log_2(10^9)\cdot 10^9 \approx 7$ GB. To reduce circuit description size, the compiler proposed in [46], called PCF (Portable Circuit Format), does not unroll the loops in the circuit until the GC protocol runs, and therefore seems to have a better scalability than the other compilers. As we elaborate in related work (see Section VIII), the existing approaches, including the

DOI 10.1109/SP.2015.32

above proposals, have certain limitations when it comes to real implementation.

A. Our approach

Our approach, TinyGarble, is based on synthesizing and optimizing circuits for the GC protocol as sequential circuits while leveraging powerful logic synthesis techniques with our newly introduced custom-libraries.

Our solution simply views the circuit generation for GC as an atypical logic synthesis task that, if properly defined, can still be addressed by conventional hardware synthesis tools. By posing the circuit generation for Yao's protocol as a hardware synthesis problem, TinyGarble naturally benefits from the elegant algorithms and powerful techniques already incorporated in existing logic synthesis solutions, see, [6], [20], [55], [66]. This view provides a radically different perspective on this important problem in contrast to the earlier work in this area that attempted to generate circuits by building new libraries for general purpose languages such as Java [36], [53], custom compilers such as [23], [46], or introduction of new programming languages such as [54], [63].

TinyGarble introduces new techniques for minimizing the number of non-XOR gates which directly results in reduced computation and communication required for the GC protocol. We do so by integrating the cost function in the new custom libraries that we design and use within our logic synthesis flow. This way, we are able to gain up to 80% improvement in the number of non-XOR gates for benchmark circuits compared to PCF [46]. The TinyGarble methodology is automated, i.e., the savings can be achieved for many functions synthesized by our method, regardless of their sophistication.

One significant contribution of TinyGarble, which differentiates it from the previous work, is expressing the function in a very compact format, namely as a sequential logic. The earlier work in this area mainly described functions in a combinational format, where the value of the output is determined entirely by the circuit inputs. This input/output relationship can be expressed by a (combinational) Boolean function and a directed acyclic graph (DAG) of binary gates. The sequential circuit description, on the other hand, allows having feedback from the output to the input by adding the notion of a state (memory). At each *sequential cycle*, the output of the circuit is determined by the current state of the system and the input. For each particular sequential cycle, the relationship between the output and the inputs for the given states can be determined as a Boolean combinational logic.

The only previous work we are aware of which implicitly hinted at the possibility of having a more compact representation is PCF [46]. It does so by embracing loops and unrolling them only at runtime. A sequential circuit, however, goes far beyond the loop embracing performed at the software level. Not only does TinyGarble embrace the high-level loops, it also enables the user to further compact the functions by folding the implementation up to its basic elements. For example, using TinyGarble, user can compress the 1024-bit addition function into only a 1-bit adder.

An important advantage of our sequential representation is providing a new degree of freedom to the user to fold the functions to simpler computing elements; i.e., the user has the freedom to choose the number of sequential cycles needed for evaluation of the function–the size of the combinational logic path between the states/inputs and the outputs. The number of gates in the sequential circuit can be managed by varying the number of cycles. The memory footprint of the GC operation is directly related to the number of gates in the sequential circuit; at any moment during garbling, only the information corresponding to the current cycle needs to be stored. Compact sequential circuits yield a small enough memory footprint that can fit mostly on a typical processor cache. This helps us to avoid costly cache misses while accessing the wire tokens during the GC protocol. Indeed, TinyGarble can enable practicable embedded implementations with a small memory footprint.

The sequential representation enables, for the first time, implementation of a universal processor for private function evaluation where the function is known only to one party. We reduce private function SFE (PF-SFE) to general SFE where the function is known by both parties. Our implementation accepts assembly instructions of the private function as input to the GC protocol. Since a processor is inherently a sequential circuit, it was infeasible to be realized with previous GC tools.

TinyGarble accepts inputs in two different formats: a standard hardware description language (HDL), or a higher level language as long as it is compatible with the existing high level synthesis (HLS) tools, e.g., the C language for SPARK [30] and Xilinx Vivado [19], or Python for PandA [59], that converts the high level language to an HDL. Beside user's manual optimization, TinyGarble performs various optimizations through standard HDL synthesis tools to generate an optimized *netlist*, i.e., list of gates, which is then transformed to be used with a GC protocol implementation, e.g., JustGarble [2] or Half Gates [72].

Contributions. In brief, our contributions are as follows:

- Adaption of established HDL synthesis techniques to compile and optimize a function into a netlist of gates for use in secure computation protocols.
- Creation of new custom libraries and setting objectives/ constraints to *repurpose* standard synthesis tools for minimizing the number of non-XOR gates

in a circuit.

- Introduction of sequential circuit description for achieving an unprecedented compactness in function representation and memory footprint.
- Providing a new degree of freedom to users to fold the functions into a sequential circuit. The user can achieve a small enough sequential circuit such that the memory required for its secure evaluation fits even in a typical processor cache. This helps to avoid costly cache misses and reduces the CPU time required for GC.
- Proof-of-concept implementation of benchmark functions such as multiplication, and Hamming distance demonstrates up to 5 orders of magnitude savings in memory footprint and up to 80% efficiency in minimizing the total number of non-XOR gates. Furthermore, TinyGarble enables implementation of large circuits that were not reported in earlier work, such as SHA-3.
- Implementing the first scalable emulation of a universal processor for private function evaluation where the number of instruction invocations is not limited by the memory required for garbling. This design is uniquely enabled by the TinyGarble sequential description. Our design is a secure general purpose processor based on the MIPS I instruction set that receives as inputs the private function from one party and the data from the other.

II. Preliminaries and Background

In this section, we provide preliminaries and related background on garbled circuits (Section II-A) and HDL synthesis (Section II-B).

A. Background on Garbled Circuit

Yao introduced the GC protocol for 2-party Secure Function Evaluation (SFE) in the 1980's [70]. GC is described as a circuit whose wires carry a string valued-token instead of a bit. Consider two parties, Alice and Bob, who want to evaluate a function $f(\cdot)$ without revealing their inputs to each other. The function needs to be represented as a combinational Boolean circuit. To begin with, we assume the circuit consists of a single gate with two input wires, w_a, w_b and one output wire w_c. Alice knows the value of input w_a denoted by v_a and Bob knows the value of input w_b denoted by v_b. The gate is also represented by a four-entry truth table $G[v_a, v_b]$. There are two main phases in Yao's protocol. First, Alice encodes or garbles the circuit by generating garbled tables. Second, Bob evaluates the output denoted by v_c without knowing anything about v_a other than what can be deduced from the output and his own input.

The steps of Yao's approach are described below.

1) For each wire w_a, Alice selects one random bit t_a called *type* and two random $(k-1)$-bit values Y_a^0 and Y_a^1, where k is a symmetric security parameter (e.g., $k = 128$). The concatenations of the first random string and the type $X_a^0 = Y_a^0 \parallel t_a$ and $X_a^1 = Y_a^1 \parallel \bar{t}_a$ are called token for semantic bit 0 and 1 respectively.
2) For each gate, Alice symmetrically encrypts the respective output tokens with the four possible combinations of the input tokens. The resulting table of ciphertexts is called *garbled table*.
3) Alice sends to Bob the garbled tables and the token corresponding to her input value.
4) Bob obliviously receives the tokens corresponding to his input through oblivious transfer (OT) [62].
5) Bob decrypts the corresponding entry in the garbled table based on the received input tokens and gets the output token.
6) Finally, Alice reveals the type of the output and Bob determines its semantic value.

In general, the circuit consists of multiple gates. Yao's protocol for this case is described below.

1) Alice chooses tokens for all the wires, constructs the garbled tables for each gate and sends these to Bob along with the tokens corresponding to her inputs.
2) Bob obliviously receives the tokens corresponding to his input values through oblivious transfer.
3) Using these tokens, Bob evaluates the circuit gate-by-gate until he evaluates all gates.
4) Finally, Alice reveals the type of the outputs and Bob determines their semantic values.

We assume the honest-but-curious model as the basis for building a stronger security protocol. Generic ways of modifying GC-based protocols such that they achieve security against stronger malicious adversaries have been proposed, e.g., [48], [50].

In our implementation, we make use of state-of-the-art optimizations for garbled circuits as described below.

1) Free XOR [44]: In this method, Alice generates a global random $(k-1)$-bit value R which is just known to her. During garbling operation for any wire w_a, she only generates a token X_a^0 and computes the other token X_a^1 as $X_a^1 = X_a^0 \oplus (R \parallel 1)$. With this convention, the token for the output wire of the XOR gates with input wires w_a, w_b and output wire w_c can be simply computed as $X_c = X_a \oplus X_b$. The proof of security for this optimization is given in [44].

2) Row Reduction [58]: This optimization reduces the size of the tables for the non-XOR gates by 25%. Here, instead of generating a token for the output wire of a gate randomly, the output token is produced as a function of the tokens of the inputs. Alice generates the output token such that the first entry of the garbled table becomes all 0 and no longer needs to be sent.

3) Garbling With a Fixed-key Block Cipher [2]: This method allows to efficiently garble and evaluate

non-XOR gates using fixed-key AES. In this garbling scheme which is compatible with the Free XOR and Row Reduction techniques, the output key X_c is encrypted with the input token X_a and X_b using the encryption function $E(X_a, X_b, T, X_c) = \pi(K) \oplus K \oplus X_c$, where $K = 2X_a \oplus 4X_b \oplus T$, π is a fixed-key block cipher (e.g., instantiated with AES), and T is a unique-per-gate number (e.g., gate identifier) called *tweak*. The proof of security is given in [2].

B. Background on HDL Synthesis

HDL synthesis refers to the process of translating an abstract form of function (circuit) presentation to the gate-level logic implementation using a series of sophisticated optimizations, transformations, and mapping [6], [20], [55], [66]. An HDL synthesis tool is a computer program which typically accepts the input circuit in some algorithmic form, logic equation, or even a table, and outputs an implementation suitable for the target hardware platform. Classic commercial/open-source HDL synthesis tools accept the input functions in the HDL format, e.g., Verilog or VHDL [13], [16], [21], [29], [56], [59] but newer ones also accept high level format, e.g., C/C++ [19], [30]. The common target hardware platforms for the synthesized logic include Field Programmable Gate Arrays (FPGA), Programmable Array Logic (PAL), and Application-Specific Integrated Circuits (ASIC).

The input functions (circuits), regardless of their HDL or higher level format, can be defined by the implementer to be purely combinational logic that is fully representable by Boolean functions, or they might be sequential logic which is a more general format.

Typical practical implementations of a logic function utilize a multi-level network of logic elements. The tools translate the input to the implementation in two steps: (i) Logic minimization; and (ii) logic optimization. Logic minimization simplifies the function by combining the separate terms into larger ones containing fewer variables. The best known algorithm for logic minimization is the ESPRESSO algorithm [7]; although the resulting minimization is not guaranteed to be the global minimum, it provides a very close approximation of the optimal, while the solution is always free from redundancy. This algorithm has been incorporated as a standard logic function minimization step in virtually any contemporary HDL synthesis tool.

Logic optimization takes this minimized format, further processes it, and eventually maps it onto the available basic logic cells or library elements in the target technology. Mapping is limited by factors such as the available gates (logic functions or standard cells) in the technology library, as well as the drive sizes, delay, power, and area characteristics of each gate.

Newer generations of synthesis programs, referred to as high level synthesis (HLS) tools, accept other forms of input in a higher level programming language [12], [14], [73], e.g., ANSI C, C++, SystemC, or Python. HLS tools are also available in both open-source and commercial forms, cf. [16], [19], [59]. The limitation of the higher level languages is that the behavior of the function is typically decoupled from the timing. The HLS tools handle the micro-architecture and transform the untimed or partially timed functional code into a fully timed HDL implementation, which in turn can be compiled by a classic synthesis tool. It is well-known that the performance of the circuits resulting from automatically compiled HLS code into HDL is inferior to the performance of functions directly written in HDL. Therefore, the main driver for the development of HLS tools is user-friendliness and not performance.

III. Global Flow

The global flow of TinyGarble is shown in Fig. 1. It consists of the following four steps:

1) The input to the TinyGarble framework is a file that describes a sequential or combinational function written in an HDL like Verilog or VHDL. The function can also be written in a high level language like C/C++ and automatically translated to HDL using an HLS tool. In the sequential circuit, the degree of folding is specified by the user.
2) A standard HDL synthesis tool compiles the HDL to generate a netlist file. The synthesis tool optimizes the netlist based on the user defined objectives/constraints and a customized library.
3) The netlist is parsed and topologically sorted. If the circuit is sequential, only its combinational part is sorted. Then, the sorted netlist is saved in a format compatible with any given GC framework e.g., Simple Circuit Description (SCD) compatible with JustGarble [2].
4) The circuit description is provided to both the garbler and evaluator to securely evaluate the function by the GC framework.

Fig. 2 shows examples of files at different steps of TinyGarble's flow for the Hamming distance function. The *hamming.c* file contains the description of the function in the C language. The user inputs this function to a HLS tool to generate the corresponding description in Verilog. The resulting Verilog file is functionally similar to the *hamming.v* file shown in the figure, but it may look more complicated and be less efficient as it is generated by an automated tool. A user can also write the description directly in Verilog to have more control on the circuit and therefore a more efficient netlist. The *hamming.v* file is provided to an HDL synthesis tool along with the TinyGarble custom libraries to generate netlist *hamming_netlist.v*. The netlist describes the same

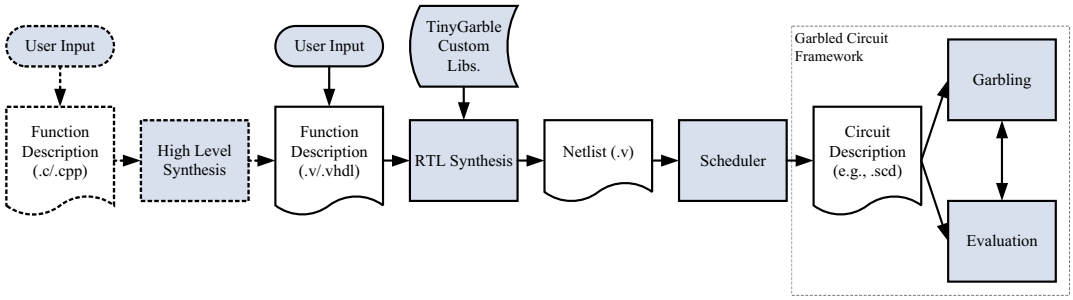


Fig. 1: Global flow of TinyGarble for both combinational and sequential synthesis. The inputs can be either a C/C++ program (translatable to HDL via a standard HLS tool) or a direct HDL description. TinyGarble is able to provide circuit description for any given GC framework.

hamming.c

```
#include <ap_int.h>
void hamming(ap_uint<1> a,
  ap_uint<1> b, ap_uint<16> &h){
#pragma HLS INTERFACE ap_none port=a,b,h

  static ap_uint<16> tmp = 0;
  ap_uint<1> c = a^b;
  if(c.get_bit(0))
    tmp++;
  h = tmp;
}
```

hamming.v

```
module hamming(clk, rst, a, b, h);
  input clk, rst, a, b;
  output reg [15:0]h;

  always@(posedge clk or posedge rst)
    if(rst)
      h <= 0;
    else
      h <= h + a^b;
endmodule
```

hamming_netlist.v

```
module hamming(clk, rst, a, b, h);
  input clk, rst, a, b;
  output [15:0]h;
...
DFF \c_reg[0] (.D(n3), .CLK(clk),
 .RST(rst), .Q(h[0]) );
...
XNOR U1 ( .A(a), .B(b), .Z(n2) );
NANDN U3 ( .A(n2), .B(n3), .Z(n3) );
...
endmodule
```

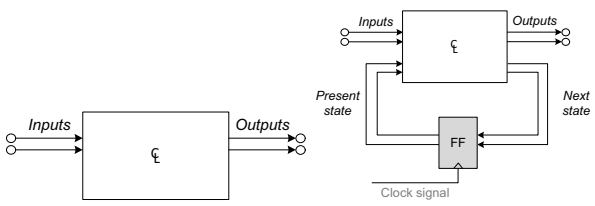
Fig. 2: Sample files at the different steps of TinyGarble's flow for Hamming distance function.

function as *hamming.c* and *hamming.v* but uses the logic cells provided in the technology library. The technology library contains 2-input-1-output logic cells to be compatible with front-end garbling tools [2], [54].

IV. Garbling and Evaluating Sequential Circuits

Sequential circuits can be used as a very compact circuit description. In the following section we first describe the concept of sequential circuits (Section IV-A) using an example and then explain the modifications required to garble/evaluate them.

A. Sequential Circuits



(a) Combinational circuit (b) Sequential circuit

Fig. 3: (a) Combinational circuit where outputs are functions of only inputs. (b) Sequential circuit where outputs are functions of inputs and present states.

Yao's GC algorithm allows secure evaluation of a Boolean circuit, i.e., an acyclic graph of binary gates (e.g., AND, OR, XOR, etc.). In digital circuit theory, such a circuit is called *combinational circuit* and defined as a memory-less circuit in which outputs are functions only of inputs, see Fig. 3a.

Another class of circuits in digital circuit theory are *sequential circuits* in which unlike in the combinational case, circuit outputs are functions of both inputs and circuit *states*. Circuit states are kept in memory elements such as Flip Flops (FF). The states can change at the end of each *clock cycle*[1].

As seen in Fig. 3b, a sequential circuit can be represented as an ensemble of a combinational circuit and feedback loops with memory elements. At each clock cycle, circuit inputs as well as the present states are fed to the combinational part. Then, it generates the outputs and next states which will be stored in the memory elements for the next cycle. The initial value of the memory elements are either a known constant value (0 or 1) or determined by an initial input value[2].

Fig. 4 demonstrates an example of a combinational and a sequential implementation for a 4-bit Adder with

[1]The clock signal oscillates between a low and a high state and its (rising) edge is typically utilized to coordinate the memory updates.

[2]In digital hardware, FF initialization is usually done by *reset* or *set* signals. In TinyGarble, we use a new signal for FF that determines the initial value. It can be connected to a constant value or input wire.

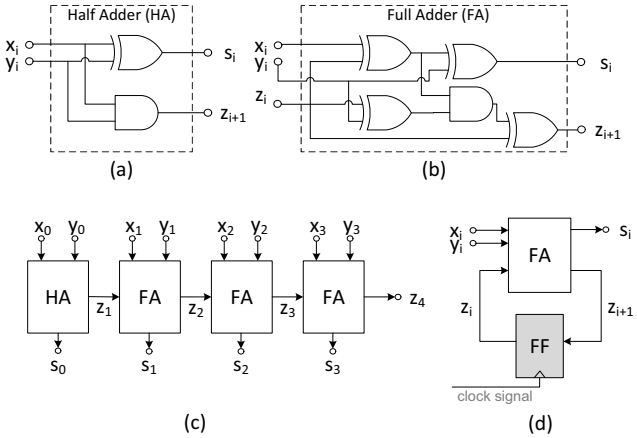


Fig. 4: Combinational and sequential design of a 4-bit Adder. (a) HA circuit. (b) FA circuit. (c) Combinational 4-bit Adder using 1 HA and 3 FAs. (d) Sequential 4-bit Adder using one FA.

inputs $x = \overline{x_3x_2x_1x_0}$ and $y = \overline{y_3y_2y_1y_0}$, producing sum $s = \overline{s_3s_2s_1s_0}$. Fig. 4a and 4b show the internal combinational circuit of a half Adder (HA) and a full Adder (FA) respectively. In Fig. 4c a combinational Adder is built by cascading 3 FAs and one HA. Fig. 4d represents a sequential implementation of a 4-bit Adder which uses one FA and a one bit FF to save the carry bit from the previous cycle. The circuit should be evaluated in 4 cycles. At the first cycle the carry bit is $z_0 = 0$. Note that, in the combinational circuit we use three FAs and one HA whereas in the sequential circuit, we have to use one FA for 4 sequential cycles. This asymmetry in the loop of Addition function introduces a very small *overhead* in GC computation and communication time as an HA circuit has fewer gates compared to a FA circuit.

However, the total number of gates for representing the function is reduced approximately by a factor of 4 when using a sequential circuit (one FA for sequential compared with three FA and one HA for combinational). This helps to limit the memory footprint for garbling and evaluation required for storing circuit description and wire tokens (k-bit per wire, see Section II-A). In a sequential circuit, the number of tokens that need to be stored in memory at any moment is proportional to the number of gates in the circuit. The wire tokens are simply over-written at each sequential cycle. Only tokens corresponding to FFs are kept for the next cycle.

Nearly all commercial circuits used in digital hardware are designed in sequential format. There are multiple reasons for preferring sequential circuit description over combinational including the reduction in complexity, area, power, and cost, as well as natural mapping of finite state machine control functions into a sequential format. Some of these reasons also provide a rationale for

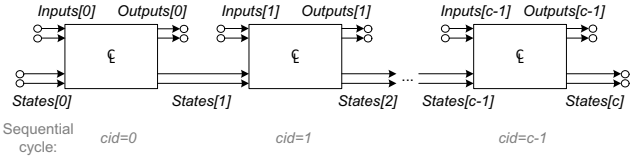


Fig. 5: Functionally equivalent unrolled sequential circuit corresponding to Fig. 3b.

sequential description of a function in GC, including: (i) reduction in size and memory footprint that is achieved by introducing the state elements and feedback loop from output to input; (ii) removing the need to perform costly compile-time/runtime loop unrolling by embracing loops within the sequential feedback loop; (iii) providing a new degree of freedom for folding by the placement of memory elements in the long combinational paths– the placement can be done in accordance with the user's objective.

During the evaluation of a sequential circuit, the combinational block is evaluated c times where c is the number of sequential cycles that the circuit operates. We can visualize this process as the unrolled combinational representation of the sequential circuit as shown in Fig. 5. The inputs of the unrolled circuit are the inputs of the combinational block in all the cycles. The same holds for the outputs, too. The present states at each cycle cid, where $0 \leq \text{cid} < c$, are equal to the next states at the previous cycle (cid $-$ 1). The present states at cid $= 0$ are equal to the input initial value.

During generation and evaluation of the garbled circuit, it must be ensured that the encryption tweaks T (see Section II-A) for each gate is unique because otherwise security is broken [32, Sect. 3.4]. In TinyGarble, to ensure the uniqueness property, we set tweak T for each gate to be the concatenation of the cycle index (cid) and the unique gate identifiers (gid) in the combinational part of the sequential circuit, i.e., $T = \text{cid}||\text{gid}$.[3] As in previous work, security and correctness of the GC garbling/evaluation follow from the uniqueness of the tweak T and the existing proofs of security and correctness, see [2], [49].

V. HDL Synthesis

As described in Section II-A, Yao's protocol requires the function to be represented as a Boolean circuit. Previous work like FairPlay [54] and WYSTERIA [63] used custom-made languages to describe a function and generate the circuit for GC operations. In our TinyGarble framework, the user may describe a function in a standard HDL like Verilog or VHDL. She may also write

[3] An alternative method would be to use a monotonic counter in the circuit generation/evaluation routines which is increased by one for each gate.

the function in a high level language like C/C++ and convert it to HDL using a HLS tool. TinyGarble uses existing HDL synthesis tools to map an HDL to a list of basic binary gates. In digital circuit theory, this list is called a *netlist*. The netlist is generated based on various constraints and objectives such that it is functionally equivalent to the HDL/HLS input function. Exploiting synthesis tools helps to reduce both number of non-XOR gates in the circuit and the garbling time while also making the framework easily accessible.

A. Synthesis Flow

In the first step, a synthesis converts functional description of a circuit into a structural representation consisting of standard logical elements. Then, it converts this structural representation into a netlist specific to the target platform. In both steps, the synthesis tool works under a set of user defined constraints/objectives like minimizing the total delay or limiting the area. In the following, we describe the details of these two steps and how we manipulate the synthesis tools in each of the steps to generate optimized netlists for SFE.

a) Synthesis library: The first step in the synthesis flow is to convert arithmetic and conditional operations like add, multiply, and if-else to their logical representations that fits best to the user's constraints. For example, the sum of two N-bit numbers can be replaced with an N-bit ripple carry adder in case of area optimization or an N-bit carry look ahead adder in case of timing optimization. A library that consists of these various implementations is called a *synthesis library*. We develop our own synthesis library that includes implementations customized for SFE. In this library, we build the arithmetic operations based on a full adder with one non-XOR gate [5] and conditional operations based on a 2-to-1 multiplexer (MUX) with one non-XOR gate [44].

b) Technology library: The next step is to map the structural representation onto a *technology library* to generate the netlist. A technology library contains basic units available in the target platform. For example, tools targeting Field Programmable Gate Arrays (FPGAs) like Xilinx ISE or Quartus contain Look-Up Tables and Flip Flops in their technology libraries, which form the architecture of an FPGA. On the other hand, tools targeting Application Specific Integrated Circuits (ASICs) like Synopsys DC, Cadence, and ABC, may contain a more diverse collection of elements starting from basic gates like AND, OR, etc., to more complex units like FFs. The technology library contains logical descriptions of these units along with performance parameters like their delay and area. The goal of the synthesis tool in this step is to generate a netlist of library components that best fit the given constraints. For HDL synthesis, we use tools targeting ASICs as they allow more flexibility in their input technology library. We design a custom technology library that contains 2-input gates as required by the front-end GC tools. We set the area of XOR gates to 0 and the area of non-XOR gates to a non-0 value. By choosing area minimization as the only optimization goal, the synthesis tool produces netlists with the minimum possible number of non-XOR gates.

An additional feature of our custom technology library is that it contains non-standard gates (other than basic gates like NOT, AND, NAND, OR, NOR, XOR, and XNOR) to increase flexibility of mapping process. For example, the logical functions $F = A \wedge B$ and $F = (\neg A) \wedge B$ requires equal effort in garbling/evaluation. However by using only standard gates, the second function will require two gates (a NOT gate and an AND gate) and store one extra token for $\neg A$ in the memory. We include four such non-standard gates with an inverted input in our custom library.

For synthesis of sequential circuits, the technology library includes memory elements. These elements can be implemented as FFs which are connected to a clock signal. Although in conventional ASIC design FFs are typically as costly as four AND gates, in our GC application, FFs do not have any impact on the garbling/evaluation process as they require no cryptographic operations. Therefore, we set the area of FFs to 0 to show its lack of impact on computation and communication time of garbling/evaluation. Moreover, we modify our FFs such that they can accept an initial value. This helps us remove extra MUXs in standard FF design for initialization.

B. Offline Circuit Synthesis

In TinyGarble, we use HDL synthesis tools in an offline manner to generate a circuit for a given functionality. This offline synthesis followed by a topological sort provides a ready-to-use circuit description for any GC framework. This approach, unlike online circuit generation, does not require misspending time for circuit generation during garbling/evaluation. It also enables the use of beneficial synthesis optimization techniques that were previously infeasible for online generation. Moreover, the synthesis tools have a global view of the circuit, unlike previous work that manually optimized small modules of the circuit. This allows more effective optimization for any arbitrary function and set of constraints.

However, the offline approach has certain limitations when it comes to generating circuits for extremely large functions. Fortunately, the sequential description helps to overcome most limitations as it generates more compact circuits. Sequential circuits are radically smaller than combinational ones with the same functionality. This property allows synthesis tools to perform more effective circuit optimization. Moreover, the compatibility of our sequential descriptions with standard synthesis tools

simplifies the workflow of circuit generation for SFE applications.

VI. Private Function Evaluation

Two-party Private Function SFE (PF-SFE) allows secure computation of a function $f_{Alice}(\cdot)$ held by one party (Alice) operating on another party's data x_{Bob} (Bob) while both the data and the function are kept private. This is in contrast to the usual setting of SFE where the function is known by both parties. PF-SFE is especially useful when the function is proprietary or classified.

It is well known that PF-SFE can be reduced to regular SFE by securely evaluating a Universal Circuit (UC) [65]. UC is a circuit capable of simulating any circuit (function) $f(\cdot)$ given the description of $f(\cdot)$ as input [45], [68]. More formally:

$$UC(f_{Alice}(\cdot), x_{Bob}) = f_{Alice}(x_{Bob}).$$

Secure evaluation of UC completely hides the functionality of $f(\cdot)$, including its topology. Subsequent works have shown how to allow PF-SFE while avoiding the overhead of UCs [41], [57].

A UC is similar to a Universal Turing Machine (UTM) [34], [67] that receives a Turing machine description $f_{Alice}(\cdot)$ and applies it to the input data (x_{Bob}) on its tape [15]. One party provides the machine description and the other one provides the initial data. The output $f_{Alice}(x_{Bob})$ resides on the tape after the operation is completed. A general purpose processor is a special realization of a UTM. It receives a list of *instructions* $f_{Alice}(\cdot)$ and applies them to the input data x_{Bob} in memory.

A. Arithmetic Logic Unit

The core of conventional processors is the Arithmetic Logic Unit (ALU) which receives two *operands* and an *opcode* indicating the desired operation. ALU supports an operation set consisting of operations like addition, multiplication, XOR, etc. The ALU circuit consists of multiple sub-circuits for these operations and a MUX which selects one of their outputs. Secure evaluation of an ALU, where the opcode comes from one party and operands come from the other party, keeps the operations private. Thus, ALU can be thought of as an emulator of a simple UC in which the input function $f_{Alice}(\cdot)$ is limited to a single operation.

One can combine a number of ALUs to make a more comprehensive UC that can support functions consisting of multiple operations. Unfortunately, this approach is not practical as the complexity of the circuit grows linearly with the number of operations. On the other hand, in conventional processors, ALUs are combined with arrays of FFs, a.k.a., *registers*, in order to store the intermediate values for supporting functions with arbitrarily large number of operations. Since none of the earlier implementations of GC explicitly supported memory elements such as FFs, the ways to connect the feedback loop around the ALU were rather limited. However, an explicit sequential description supported by TinyGarble allows us to leverage conventional processor architectures. Therefore, the TinyGarble methodology not only provides a powerful method for generating compact circuits with a low overhead for SFE, but also paves the way for systematically building scalable sequential circuits that can be used for PF-SFE.

The idea of using an ALU or a *universal next-instruction circuit* in the GC protocol can also be found in [51]. The objective of that paper was improving efficiency of SFE where the function is known by both parties, unlike PF-SFE where the function is private. Nonetheless, instead of ALU they eventually decided to use an *instruction-specific circuit* which leaks information about the function but results in less effort for non-private function evaluation.

B. Memory

The processor accesses the memory while executing an instruction to read the instruction and data and write the data back. If the memory is securely evaluated along with the processor, the access patterns must be also oblivious to both parties. On the other hand, if the memory is not evaluated securely, the access patterns could be revealed that in turn could reveal information about the function to Bob and about the data to Alice. For example, the instruction read pattern discloses the branching decisions in the function which may leak information about the data. Because of TinyGarble sequential methodology, the memory can be easily implemented using MUX and arrays of FFs. Thus, it can be included in the processor circuit to be evaluated securely using the GC protocol. However, inclusion of MUXs and FFs increases the operation time and communication linearly with respect to the memory size.

One alternative approach for hiding memory access patterns is the use of Oblivious Random-Access Machine (ORAM) protocols [27] which allows oblivious load/store operations with amortized polylogarithmic overhead [25], [28], [51], [52]. For the sake of simplicity, we do not use ORAM in this work. However, one can simply connect our implementation of PF-SFE to an ORAM to benefit from its lower amortized complexity. As another alternate, [71] showed that algorithms can sometimes be rewritten to use data structures such as stacks, queues, or associative maps for which they give compact circuit constructions of poly-logarithmic size.

C. Secure Processor

We assume Alice provides the private function $f_{Alice}(\cdot)$ and Bob provides private data x_{Bob}. At the end of the operation, only Bob learns the output $f_{Alice}(x_{Bob})$. Note that we are not considering the case where both parties learn the output as that would allow Alice to learn Bob's private data with an identity function ($f \equiv I$). The protocol is as follows:

1) Alice and Bob agree on an instruction set architecture (ISA), its implementation (i.e., the processor circuit), the maximum number of sequential cycles, and the configuration of data x_{Bob} in the memory.
2) Alice compiles the function $f_{Alice}(\cdot)$ according to the ISA. Her input is the compiled binary of the function.
3) Bob prepares his input based on the agreed configuration to initialize the processor memory.
4) Using any secure GC framework, Alice garbles the processor circuit for the maximum number of sequential cycles and Bob, after receiving his inputs with OT, evaluates the garbled processor circuit for the same number of cycles.
5) Alice reveals the output types such that Bob learns the value of the output $f_{Alice}(x_{Bob})$ stored in memory. This needs to be done only for agreed memory locations containing the outputs such that Bob does not learn intermediate values in the memory.

Because of secure evaluation using the GC protocol in Step 4, no information about values in the circuit will be leaked except the output. Without knowing internal values in the processor circuit, none of the parties can distinguish instructions or memory access patterns. In the following, we demonstrate an implementation of a processor supporting the MIPS (Microprocessor without Interlocked Pipeline Stages) ISA, as an example of a garbled processor for securely evaluating private functions.

D. MIPS

MIPS is a text-book Reduced Instruction Set Computing (RISC) ISA [40]. The RISC ISA consists of a small set of simplified assembly instructions in contrast to Complex Instruction Set Computing (CISC) (e.g., x86 ISA) which includes more complex multi-step instructions [33]. We choose a RISC ISA processor instead of CISC for the following main reasons: (i) lower number of non-XOR gates, (ii) simple and straightforward implementation, and (iii) availability and diversity of open-source implementations. Moreover, we choose a single-cycle MIPS architecture (i.e., one instruction per sequential cycle). Other architectures (i.e, multi-cycle and pipelined) increase the performance of the processor by parallelization. However, the GC protocol does not benefit from such low level parallelization. The only important factor for GC is the total number of non-XORs which is smaller in the single-cycle MIPS. We follow the Harvard Architecture which has distinct instruction memory (IM) and data memory (DM) in order to separate the parties' inputs. IM is a Read-Only Memory (ROM) that stores Alice's instructions. DM is a Random Access Memory (RAM) that is initialized with Bob's input. The parties' inputs are connected to the initial signal inputs of FFs in the memories. Bob's outputs are connected to the outputs of FFs in the specified address of DM. The output address in DM is part of the agreed memory configuration.

Fig. 6 shows the overall architecture of our 32-bit MIPS processor. It is based on the Plasma project in opencores [64]. We modified the circuit such that the instruction ROM (IM) and the data RAM (DM) are separated. The original Plasma processor supports all the MIPS I ISA except unaligned memory access. In our implementation, we also omit division instructions because of their large overhead. Any arbitrary C/C++ function can be easily compiled to MIPS I assembly code using a cross-platform compiler e.g., GNU gcc.

In 32-bit MIPS, the program counter (*PC*) is a 32-bit register (array of FFs) that points to the instruction being executed at the current cycle. The instruction is fetched from IM based on the current PC value. The *controller* unit is responsible for setting signals to perform the instruction. In 32-bit MIPS, the *register file* consists of 32 registers of 32-bit each. In each cycle, at most two registers can be read and at most one register can be written back. ALU receives the read register(s) or a sign extended *immediate* as operands. ALU also receives an opcode from the controller unit. The output of ALU will be either written back to the register file or fed to DM as an address for load/store. The loaded data from DM is written back to the register file. In each cycle, PC is incremented by 4 to point to the next instruction in IM or is changed according to a branch or jump instruction.

VII. Evaluation

We use a variety of benchmark functions to evaluate the performance and practicability of TinyGarble. In this section, we first describe our experimental setup (Section VII-A) and metrics for quantifying the performance of TinyGarble (Section VII-B). We outline the performance comparison of TinyGarble (with HDL synthesizer and our custom libraries) on combinational benchmark functions with PCF [46], one of the best known earlier automated methodologies to generate circuits for garbling in Section VII-D. TinyGarble's performance in generating sequential circuits for benchmark functions using a standard HDL synthesis tool is demonstrated in Section VII-E. Section VII-F shows the CPU time for various numbers of sequential cycles which demonstrates the effect of memory footprint reduction in garbling time.

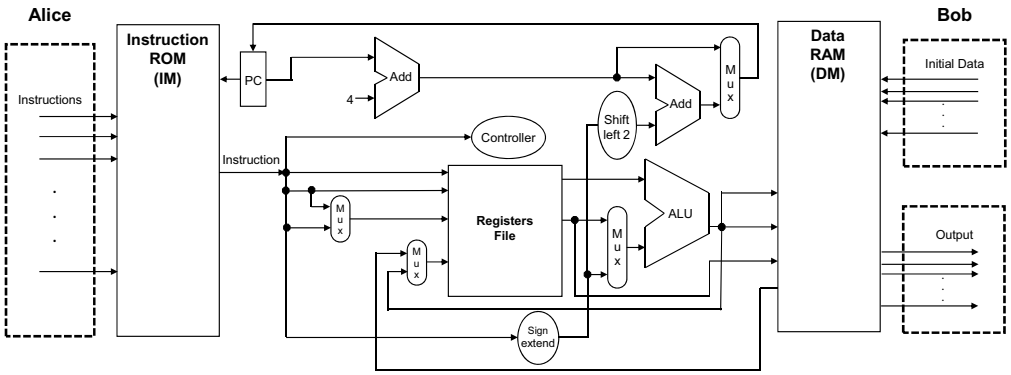


Fig. 6: Lite MIPS architecture. Alice's and Bob's inputs and the output are shown.

Section VII-G shows a comparison between TinyGarble's performance using an HLS tool (input written in C) and using a conventional HDL synthesis tool (input given in Verilog). Lastly, Section VII-H shows the result of our garbled processor and implementation of Hamming distance as a benchmark.

We also compare the performance of the commercial logic synthesis tool with the academic, open-source tools in Appendix A. We show that in most cases, the performance of the open-source tool is comparable to the commercial tool.

A. Experimental Setup

The circuit generations are all done on a system with Linux RedHat Server 5.6, 8 GB of memory, and Intel Xeon X5450 CPU @ 3 GHz. We use another system with Ubuntu 14.10 Desktop, 12.0 GB of memory, and Intel Core i7-2600 CPU @ 3.4GHz to assess the timing performance of the sequential garbling scheme in Section VII-F.

Two sets of HDL synthesis tool chains are used in our experiments: one commercial and one open-source (Appendix A). Our commercial HDL level synthesis tool is Synopsis Design Compiler (DC) 2010.03-SP4 [13]. We also use the Synopsis Library Compiler from the DC package to interpret our custom technology library. In Section VII-G, we utilize Xilinx Vivado HLS [19], a commercially available HLS tool whose inputs are written in the C/C++ programming language. We emphasize that TinyGarble can operate with any commercial or open-source sequential HDL-level (or HLS) synthesizer, as long as the synthesizer is capable of performing state-of-the-art logic optimization and mapping algorithms.

B. Performance Metrics

We use the following metrics to measure the efficiency of TinyGarble for generating garbled circuits:

- *Memory Footprint Efficiency* (MFE):

$$MFE = \frac{q_0}{q},$$

where q_0 is the total number of gates in the reference circuit and q is the total number of gates in the circuit under evaluation. The maximum number of tokens that need to be stored at any point during garbling/evaluation as well as memory required for storing circuit description is directly proportional to the number of gates in both sequential and combinational circuits. Thus, the total number of gates is approximately proportional to the memory footprint.

- *Number of Garbled Tables* ($\#GT$):

$$\#GT = \#nonXOR \times c,$$

where $\#nonXOR$ is the number of non-XOR gates in a circuit and c is the number of sequential cycles that the circuit needs to be garbled/evaluated. In free XOR-based GC schemes, each non-XOR gate requires a garbled table to be generated by the garbler and sent to the evaluator at each sequential cycle. Thus, this metric is an estimate of both the computation and communication time.

- *Garbled Tables Difference* (GTD (%)):

$$GTD = \frac{\#GT - \#GT_0}{\#GT_0} \times 100,$$

where $\#GT_0$ is the total number of garbled tables for the reference circuit and $\#GT$ is the total number of garbled tables for the circuit under evaluation. When comparing a sequential with a combination circuit, positive GTD shows an *overhead* (caused by folding a circuit with an asymmetric loop, see Section IV) in total computation and communication time resulting from an excessive number of garbled tables generated in the sequential circuits. However, in general, negative GTD shows improvement in the number of non-XOR gates and generated garbled tables that results from logic synthesis optimization.

C. Benchmark functions

We evaluate TinyGarble's circuit generation method on various benchmark functions. Several of these functions have been used in previous works, e.g., PCF [46]. In the following, we introduce our benchmarks and explain how we fold them into a sequential representation.

Sum. This function receives two N-bit inputs and outputs an N-bit sum. The sum function is implemented in N steps of one bit sums by keeping the carry bit. Thus, it can be folded up to N times without any significant overhead in number of garbled tables ($\#GT$).

Hamming Distance. This function receives two N-bit inputs and outputs the $\log_2(N)$-bit Hamming distance between them. The Hamming distance between two numbers is the number of positions at which the corresponding bits are different. A possible combinational implementation of the N-bit Hamming distance uses a binary tree of adders that sums all 1-bit values from the bit differences to a final Hamming distance consisting of $\log_2(N)$ bits [5]. This implementation cannot be folded easily. However, we can fold this function into N-cycles of one XOR and one $\log_2(N)$-bit adder. This causes an overhead compared to the combinational circuit.

Compare (Millionaires problem). This function receives two N-bit unsigned input values and outputs a greater than signal consisting of one bit that indicates if the first input is greater than the second one. The comparison function can be implemented in N steps of subtraction by keeping the carry bit [43]. Thus, it can be folded up to N times without any significant overhead.

Multiplication. This function receives two unsigned N-bit inputs and outputs their unsigned N-bit product. The multiplication function consists of N additions and shifts. The shift operations result in an asymmetric structure in this function. Thus, folding it up to N times may increase the overhead.

Matrix Multiplication. This function receives two $N \times N$ matrices consisting of 32-bit unsigned numbers and outputs an $N \times N$ matrix equal to the product of the input matrices. The $N \times N$ matrix multiplication function consists of three N-cycle nested loops with a symmetric structures. It can be folded up to N^3 times without any significant overhead.

AES-128. This function receives a 128-bit plaintext and 128-bit round keys and outputs a 128-bit ciphertext based on the Rijndael algorithm. The AES-128 function consists of 10 rounds with almost symmetric structure. Ideally, it can be folded up to 10 times without any significant overhead.

SHA3. This function receives 576-bit inputs and outputs a 1600-bit number equal to the SHA3 hash of the input. We implement the Keccak-f permutations[1600] procedure for realizing this function. The SHA3 function consists of 24 steps, each with a symmetric structure. It can be folded 24 times without any significant overhead.

D. Combinational Garbled Circuit

To show the performance gain of using our custom libraries, we compare TinyGarble combinational circuits with circuits reported in PCF [46]. We choose PCF because among the *automated* GC tools available at the time of writing, it shows better results for most of the benchmarks. In some other work like FastGC [36], a number of benchmark circuits have been more aggressively improved (compared to PCF) using ad-hoc and mostly manual optimizations, but without a generalizable methodology.

The comparison is shown in Table I. We compute the garbled tables difference GTD (see Section VII-B) of various benchmarks by using circuits reported in PCF as reference (GTD^{PCF}). It can be seen that the combinational circuits generated by TinyGarble have non-positive GTD^{PCF} which means that the number of garbled tables are less than or equal to that of PCF circuits. We also compare the memory footprint by computing the memory footprint efficiency MFE with PCF as reference (MFE^{PCF}). We observe that MFE^{PCF} is larger than 1 (up to 9.3). This means that even without using sequential circuits, the memory footprint can be reduced by almost an order of magnitude by using TinyGarble custom libraries and standard HDL synthesis.

In case of Hamming distance, TinyGarble shows, on average, 80% improvement in number of garbled tables. Another automated tool CBMC-GC [23] reports better result compared to PCF for Hamming 160 (non-XOR $4{,}738$, total gates $20{,}356$). However, TinyGarble shows 66% improvement in number of garbled tables compared to CBMC-GC. In case of 256-bit and 1024-bit Multiplication, and 8×8 and 16×16 Matrix Multiplication, because of the huge (impractical) sizes, Synopsis DC was unable to generate the entire combinational circuit. This is because Synopsis DC is a tool developed for commercial applications. The real-life applications are almost always written sequentially, otherwise the design would not be scalable or even amenable to offline compilation onto a hardware circuit. We emphasize that our sequential circuit ($c > 1$) provides the exact same functionality while having a very small memory footprint compared with the reference circuit.

Comparison with Hand-Optimized Circuits: The netlists generated by the automated flow of TinyGarble show similar performance as the hand-optimized netlists in many cases. For example, [43] describe an N-bit sum circuit with $5N$ gates of which N gates are non-XOR and an N-bit comparison circuit with $4N$ gates of which N gates are non-XOR. The circuits generated by TinyGarble have about the same number of gates for these two functions. Note that one can always add

any hand-optimized module to the synthesis library of TinyGarble.

E. Sequential Garbled Circuit

As described in Section IV, the user has the degree of freedom to fold a combinational circuit and convert it to a sequential one to reduce the memory footprint. c denotes the number of sequential cycles required to garble/evaluate the circuit. This value demonstrates the amount of folding that is performed before the circuit is input to the synthesizer. The user defines the value of c and writes her own input function in an HDL or a higher level language such that the function is evaluated in c sequential cycles.

We use Memory Footprint Efficiency (MFE), to evaluate the reduction in memory requirement. We use TinyGarble combinational circuits ($c = 1$) as reference. The ideal MFE for a circuit with c sequential cycles is c. We also compare the memory footprints of sequential circuits with combinational circuits reported in PCF (MFE^{PCF}).

As explained in Section IV-A, the folding process may introduce some overhead on the total number of garbled tables. To assess this overhead, we compute the Garbled Tables Difference (GTD) of the sequential circuit using TinyGarble combinational circuits as reference. The ideal GTD is 0%, which means that the total number of garbled tables should be equal to those for a functionally equivalent combinational circuit. We also compare the number of garbled tables of sequential circuits with combinational circuits reported in PCF (GTD^{PCF}) to show that even with the incurred overhead, the number of garbled tables for sequential circuits is still less than that of PCF for most cases.

Table II shows the number of total gates, non-XOR gates, MFE, GTD, MFE^{PCF}, and GTD^{PCF} of the benchmark circuits for various input widths. MFE, GTD are computed with TinyGarble combinational circuits (with $c = 1$) as reference. MFE^{PCF}, and GTD^{PCF} use the circuits reported in PCF as reference. In the case of AES 128, we compare our implementation with the manually optimized circuit reported in FastGC [36] because PCF did not report it directly.

We provide a few highlights from Table II. TinyGarble is able to decrease the size of the sum of two 1024-bit numbers by $1{,}022.8$ times (i.e., more than three orders of magnitude) without affecting the number of garbled tables (GTD) compared with its own combinational circuit. For Hamming 16000, TinyGarble is able to decrease the memory footprint by $7{,}345.5$ times (i.e., about 4 orders of magnitude) while reducing the number of garbled tables by 47.3% in comparison with the circuit reported in PCF. In case of Mult 1024, TinyGarble shrinks the memory footprint by a factor of $2{,}504.4$ while reducing the number of garbled tables by 79.4% when compared with the result in PCF. For a 16×16 matrix multiplication, a $4{,}434.1$ more compact TinyGarble solution with 6% less garbled tables compared with PCF is available. By folding AES-128 10 times, the total number of gates is reduce by a factor of 13.9 compared to the FastGC circuit without any overhead in the number of non-XORs. Observe that the savings are typically more for larger bit-widths while extreme foldings can introduce an increased overhead in number of garbled tables due to the resulting asymmetry.

Because of the TinyGarble superior scalability, we are able to implement functions that have never been reported before, such as SHA-3, which can be represented using $344{,}059$ and $6{,}788$ gates respectively.

F. Effect of Folding on Garbling Time

So far, we have only reported the overhead in terms of garbled tables (GTD) that is a function of the number of non-XOR gates. As explained in [2], if we see garbling as a cryptographic primitive, its computation time (without considering communication) will also be interesting. In practice, smaller circuits which can fit entirely in the processor cache result in fewer cache misses and therefore, consume less CPU cycles for garbling. To better observe the impact of cache speed-up for the compact circuits resulting from TinyGarble, Fig. 7 depicts the CPU Time (left y-axis) and the memory footprint of wire tokens (right y-axis) versus c (x-axis) for the $32{,}768$-bit Sum function. As mentioned earlier, the memory footprint is directly proportional to the total number of gates in the sequential circuit.

This experiment is done using our sequential garbling scheme based on JustGarble [2] that includes using Free XOR, Row Reduction, and Fixed-key AES garbling techniques (see Section II-A). We use an Intel Core i7 CPU @ 3.40GHz which supports the AES-NI instruction set. The CPU cycle is measured as the average of $10,000$ trials using RDTSC instruction. For security parameter $k = 128$ (the bit-width of wire token, see Section II-A), we store 128-bit per tokens. For garbling in JustGarble, we store 2 tokens, 2 32-bit input indexes, and an 8-bit gate-type per gate. Thus, the memory footprint is approximately 328-bit per gate in garbling operation. Folding the circuit by a factor of $c \in [1 : 32{,}768]$ constantly decreases the memory footprint while the computation effort remains almost constant. Interestingly, as can be seen from the figure, the number of CPU cycles sharply decreases by $1.6\times$ just when we fold four times ($c = 4$) compared to $c = 1$. This is because for $c \geq 4$, the memory space required for garbling completely fits in the cache. The minimum CPU cycle per gate happens at $c = 2{,}048$ for 3.2 KB memory footprint. This signifies the fact that even for large functions, we can use the sequential approach to fit the corresponding memory

TABLE I: Comparison of TinyGarble combinational circuits with PCF. In case of AES 128, the result is compared with FastGC.

Function	PCF (*FastGC)		TinyGarble: Combinational			
	Non-XOR	Total gates	Non-XOR	Total gates	GTD^{PCF}	MFE^{PCF}
Sum 128	345	1,443	127	634	-63.2%	2.3
Sum 256	721	2,951	255	1,274	-64.6%	2.3
Sum 1024	2,977	11,999	1,023	5,114	-65.6%	2.3
Hamming 160	880	4,368	158	1,039	-82.0%	4.2
Hamming 1600	6,375	32,912	1,597	10,679	-74.9%	3.1
Hamming 16000	97,175	389,312	15,994	107,226	-83.5%	3.6
Compare 16384	32,229	97,733	16,384	65,536	-49.2%	1.5
Mult 64	24,766	105,880	3,925	11,439	-84.2%	9.3
Mult 128	100,250	423,064	16,046	47,620	-84.0%	8.9
Mult 256	400,210	1.66E+06	-	-	-	-
Mult 1024	6.37E+06	2.56E+07	-	-	-	-
MatxMult 3x3	27,369	92,961	27,369	91,305	0.0%	1.0
MatxMult 5x5	127,225	433,475	127,225	425,775	0.0%	1.0
MatxMult 8x8	522,304	1.78E+06	-	-	-	-
MatxMult 16x16	4.19E+06	1.43E+07	-	-	-	-
AES 128	5,760*	36,048*	5,760	29,811	0.0%	1.2
SHA3 1600	-	-	38,400	160,054	-	-

TABLE II: Comparison of TinyGarble sequential circuits with PCF and TinyGarble combinational circuits. In case of AES 128, the result is compared with FastGC.

Function	TinyGarble: Sequential						
	c	Non-XOR	Total gates	GTD^{PCF}	MFE^{PCF}	GTD	MFE
Sum 128	128	1	5	-62.9%	288.6	0.8%	126.8
Sum 256	256	1	5	-64.5%	590.2	0.4%	254.8
Sum 1024	1,024	1	5	-65.6%	2,399.8	0.1%	1,022.8
Hamming 160	32	10	41	-63.6%	106.5	102.5%	25.3
Hamming 1600	320	13	47	-34.7%	700.3	160.5%	227.2
Hamming 16000	3,200	16	53	-47.3%	7,345.5	220.1%	2,023.1
Compare 16384	16,384	1	4	-49.2%	24,433.3	0.0%	16,384.0
Mult 64	16	316	561	-79.6%	188.7	28.8%	20.4
Mult 128	32	636	1,137	-79.7%	372.1	26.8%	41.9
Mult 256	64	1,276	2,539	-79.6%	653.7	-	-
Mult 1024	256	5,116	10,219	-79.4%	2,504.4	-	-
MatxMult 3x3	27	961	3,227	-5.2%	28.8	-5.2%	28.3
MatxMult 5x5	125	961	3,227	-5.6%	134.3	-5.6%	131.9
MatxMult 8x8	512	961	3,227	-5.8%	552.4	-	-
MatxMult 16x16	4,096	961	3,227	-6.0%	4,434.1	-	-
AES 128	10	576	2,588	0.0%	13.9	0.0%	11.5
SHA3 1600	24	1,668	6,788	-	-	4.3%	23.6

space requirement into the cache and avoid the penalty of cache misses, thus achieving a large reduction in garbling time.

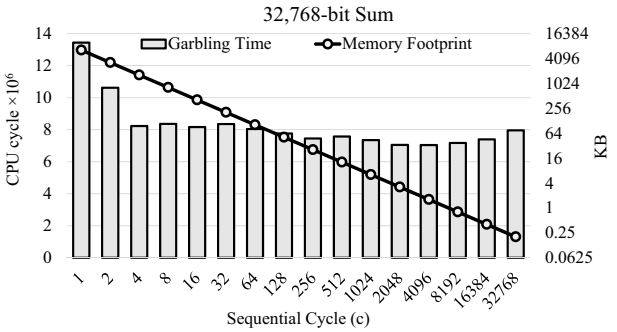


Fig. 7: Garbling 32,768-bit Sum function. The CPU time in number of cycles and the approximate memory footprint in KBytes (y-axis) versus c (x-axis) are shown.

G. High Level Synthesis Tools

The design automation community has been working on tools that work with higher-level languages and abstractions than HDL. While a host of commercial and academic HLS tools are available [16], [19], [30], [59], we selected the Xilinx Vivado HLS for compiling C code to HDL which can then be synthesized using a conventional HDL synthesis tool. The HLS engine in the Vivado suite is built upon the xPilot project [73].

Table III demonstrates a comparison between the performance of the circuits generated using C input to the HLS tool (C→Verilog) and a direct Verilog input. As can be seen from the table, the resulting memory footprint could increase by a factor between 1 and 4, while the number of garbled tables varies in a range of 3 to 9 times. It is well known that writing the HDL level code which contains the time information and more detailed structural/behavioral description would yield much more efficient circuits than the code written in a higher level language.

H. Evaluation of MIPS

We implement general purpose processor for PF-SFE using MIPS I where one user provide function description in assembly and the other provides the data. Support of sequential circuits in TinyGarble enables us to use the MIPS circuit description in Plasma project [64] without major modifications. In the following, we provide the result of MIPS implementation and its memory footprint and communication load. Lastly, we present implementation of Hamming distance with variable input length as a benchmark of private function application on MIPS.

1) MIPS Implementation: We used TinyGarble to generate the netlist for the MIPS sequential circuit. Table IV shows the total number of gates and non-XOR gates for each module of the MIPS processor with 64×32bit DM and IM. The sum of non-XORs for each module is $14{,}997$. However, when the modules are combined together to form the entire MIPS processor, the synthesizer optimizes the circuit such that the total number of non-XORs is reduced by 14.95% to $12{,}755$. The memory footprint for storing tokens during garbling MIPS is approximately the size of two tokens times the total number of gates which is $2 \times 128 \times 31{,}719$bit $= 991$ KB for token bit-width $k = 128$. The communication load between parties for invocation of one instruction (one sequential cycle) is approximately the size of three tokens times the number of non-XOR gates which is $3 \times 128 \times 12{,}755$bit $= 598$ KB with Row Reduction optimization.

TABLE IV: Number of total gates and non-XOR gates in the MIPS implementation. The global optimization of TinyGarble reduces the overall number of gates compared to that of the sum of individual modules.

Modules	Total gates	Non-XOR
Controller	509	470
Bus	603	590
ALU	651	346
Shifter	1,362	1,092
Mult	2,147	1,792
Reg File	8,880	3,023
IM	6,048	2,016
DM	13,779	5,423
PC	309	245
Total	34,288	14,997
MIPS	31,719	12,755
Global optimization	7.49%	14.95%

2) Benchmark: Hamming Distance: We implemented the Hamming distance function as a proof-of-concept for our secure MIPS. It counts the number of different elements in two arrays A and B with variable length l. For the hand-optimized assembly code shown in Fig. 8, the function requires at most $7 + 9l$ sequential cycles (instructions) to evaluate. Thus, based on Table IV, this function requires overall $12{,}755 \times (7 + 9l)$ non-XOR gates. It has only 16 instructions and is stored in 16×32bit of the IM. The function requires that l, A, and B are stored in addresses 0, $[2 : l+1]$, and $[l+2 : 2l+1]$ of DM respectively. It will store the Hamming distance of A and B in address 1.

VIII. Related Work

We classify related work into compilers for GCs (Section VIII-A), libraries for GCs (Section VIII-B), GC implementations with hardware accelerators (Sec-

TABLE III: Comparison of performance of the circuits generated using C input to HLS and a direct Verilog input to the HDL synthesizer.

Function	c	C → Verilog		Verilog		MFE	GTD
		Non-XOR	Total gates	Non-XOR	Total gates		
16384-bit Compare	1,024	51	70	16	64	0.91	219%
	4,096	15	21	4	16	0.76	275%
	16,384	6	9	1	4	0.44	500%
160-bit Hamming	1	1,264	2,142	371	1,230	0.57	241%
	32	91	146	17	35	0.24	435%
	160	45	71	10	19	0.27	350%
1024-bit Sum	1	3,067	5,115	1,023	5,114	1.00	200%
	32	195	292	32	160	0.55	509%
	1,024	9	11	1	5	0.45	800%

```
#hamming distance
#between A and B with length of l
hamming:
    lw $9, 0($0) #load l into $9
    sll $9, $9, 2 #$9 = $9*4
    addi $2, $0, 8 #$2 := A
    add  $3, $2, $9 #$3 := B = A + l
    #answer; no need to reset
    #addi  $10, $0, 0
    #l+=2 to compare with end of A
    addi  $9, $9, 8
loop: #done if A==end of A
    beq $2, $9, done
    lw $4, 0($2) #load *A
    lw $5, 0($3) #load *B
    xor $6, $4, $5 #$6==0
    beq $6, $0, same #goto A[i]!=B[i]
    addi $10, $10, 1 #answer++
same:#A++ #B++
    addi $2, $2, 4
    addi $3, $3, 4
    j loop #jump back to the top
done: #store answer
    sw $10, 4($0)
end: #while(1)
    j end
```

Hamming.s

Fig. 8: Hamming distance assembly code.

tion VIII-C), and GC implementations on mobile devices (Section VIII-D).

A. Compiler for Garbled Circuits

The following tools compile high level function descriptions into a Boolean circuit which can be used in GC. The first realization of GCs was Fairplay [54] which provides a custom high level procedural language called SFDL (Secure Function Definition Language) that is compiled into a circuit description language, SHDL (Secure Hardware Description Language). Another compiler is TASTY [31] which allows to combine garbled circuits and homomorphic encryption. The compiler of [47] for the first time showed scalability to circuits consisting of billions of gates, e.g., a 4095x4095-bit edit distance circuit with almost 6 Billion gates. The compiler of [23] allows to use a subset of ANSI C as input language.

To reduce the memory overhead for storing large circuits and hence increase scalability, PCF [46] introduced loops that, if given manually in the high level language, are kept until the GC evaluation. In contrast to PCF, TinyGarble allows to infer loops automatically and also allows to optimize across multiple sub-circuits.

B. Libraries for Garbled Circuits

Instead of compiling circuits, FastGC [36] proposed to use a library-based approach where circuits can be programmed and easily integrated into high-level applications. Another GC library is VMCRYPT [53] that allows to dynamically construct and deconstruct sub-circuits. FastGC was extended in [32] to re-use the same sub-circuits. Another library for secure computation is ABY that allows the efficient combination of multiple secure computation approaches [18].

In all these library-based approaches the circuits and their decomposition into sub-circuits has to be specified manually by the programmer, whereas we provide an automated approach.

C. GC Implementations with Hardware Accelerators

The following works provide better performance by implementing garbled circuits in hardware, on GPUs, or using AES-NI available in recent CPUs. These works can benefit from the compact representation generated by TinyGarble.

Järvinen et al. [38] proposed a generic hardware architecture for GC. They realized two FPGA-based prototypes: a system-on-a-programmable-chip with access to a hardware crypto accelerator targeting smartcards and smartphones, and a stand-alone hardware implementation targeting ASICs.

Recently, several accelerations of GCs using GPUs have been proposed. Husted et al. implemented Yao's GC by using optimizations such as Free XOR, pipelining, and OT extension [37]. Pu et al. realized dynamic programming based on GC to solve the Edit-Distance

(ED) and the Smith-Waterman (SW) problems [61]. They also used the same optimizations as [37] along with permute-and-encrypt, efficient lookup-table design, and compact circuits [61]. Frederiksen et al. implemented a secure computation protocol with security against malicious adversaries based on cut-and-choose of Yao's garbled circuit and an efficient OT extension for two-party computation on GPUs [24].

Bellare et al. propose JustGarble in which they use fixed-key AES for circuit-garbling [2]. They show their implementation using AES-NI can efficiently garble and execute a circuit far faster than any prior report.

D. GC Implementations on Mobile Devices

Our approach for generating compact circuit representations is also beneficial when performing secure computation on resource constrained devices such as mobile devices which have a limited amount of main memory. Secure computation on mobile devices using garbled circuits was proposed in [35]. Also the protocol described in [17], which uses a smartcard installed in the mobile device, can benefit from our more compact circuit representation. In [10], [11] the mobiles no longer need to process circuits any more as GC generation and evaluation is outsourced to cloud servers.

IX. Conclusion and Future Directions

We present TinyGarble, an automated tool that can generate highly compact and scalable circuits for Yao's garbled circuit (GC) protocol. We are the first to define the circuit generation for GC as a sequential synthesis problem, and to leverage the powerful and established HDL synthesis techniques with our custom-libraries and objectives. We improve the results of one of the best automatic tools for GC generation, PCF [46], by several orders of magnitude: for instance TinyGarble compacts the $1{,}024$-bit multiplication by $2{,}504$ times, while decreasing the number of non-XOR gates by 80%; we compress the $16{,}000$-bit Hamming distance by a factor of $7{,}345$ times and with 47% less non-XOR gates. Further, TinyGarble is able to implement functions that have never been reported before, such as SHA-3. We perform extensive benchmarking with both commercial and open source hardware synthesis tools and compare the results. Our approach strongly improves the existing results towards practical secure computation with many exciting applications. For instance, TinyGarble is an enabling technology for performing GC operations on mobile platforms, which is prohibitively expensive using the prior techniques. Moreover, we introduce a scalable secure processor for private function evaluation (PF-SFE). The processor is based on the MIPS architecture and the private function can be compiled using ubiquitous tool, e.g., gcc. In future work we will investigate the possibility of connecting Oblivious RAM (ORAM) to our secure processor to benefit from its lower amortized complexity for memory access. We are also working on interfacing TinyGarble with other GC schemes, e.g., the recently proposed Half Gates method [72].

Acknowledgments

The authors would like to thank Prof. David Evans for his very insightful comments, and anonymous reviewers for their helpful comments and suggestions to improve this work. The Rice University authors are partially supported by an Office of Naval Research grant (ONR-R17460), a National Science Foundation grant (CNS-1059416), and U.S. Army Research Office grant (ARO-STIR-W911NF-14-1-0456) to ACES lab at Rice University. The work of authors at TU Darmstadt is in parts 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 EC SPRIDE, and the Hessian LOEWE excellence initiative within CASED.

References

[1] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Secure evaluation of private linear branching programs with medical applications. In *ESORICS*, pages 424–439. Springer, 2009.

[2] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*, pages 478–492. IEEE, 2013.

[3] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *CCS*, pages 784–796. ACM, 2012.

[4] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS*, pages 257–266. ACM, 2008.

[5] Joan Boyar and Ren Peralta. Concrete multiplicative complexity of symmetric functions. In *MFCS*, pages 179–189. Springer, 2006.

[6] Robert K Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R Wang. MIS: A multiple-level logic optimization system. *TCAD*, pages 1062–1081, November 2006.

[7] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[8] Justin Brickell, Donald E Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *CCS*, pages 498–507. ACM, 2007.

[9] Julien Bringer, Hervé Chabanne, and Alain Patey. Privacy-preserving biometric identification using secure multiparty computation: An overview and recent trends. *Signal Processing Magazine, IEEE*, pages 42–52, 2013.

[10] Henry Carter, Charles Lever, and Patrick Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. In *ACSAC*. ACM, 2014.

[11] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Secure outsourced garbled circuit evaluation for mobile phones. In *USENIX Security*, pages 289–304. USENIX, 2013.

[12] Chi-Min Chu, Miodrag Potkonjak, Markus Thaler, and Jan Rabaey. HYPER: An interactive synthesis environment for high performance real time applications. In *ICCD*, pages 432–435. IEEE, 1989.

[13] Design Compiler. Synopsys inc. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler, 2000.

[14] Miguel R Corazao, Marwan A Khalaf, Lisa M Guerra, Miodrag Potkonjak, and Jan M Rabaey. Performance optimization using template mapping for datapath-intensive high-level synthesis. *TCAD*, pages 877–888, 2006.

[15] Martin Davis. *Engines of Logic: Mathematicians and the Origin of the Computer*. WW Norton & Co., Inc., 2001.

[16] Jan Decaluwe. MyHDL: a python-based hardware description language. *Linux journal*, page 5, 2004.

[17] Daniel Demmler, Thomas Schneider, and Michael Zohner. Ad-hoc secure two-party computation on mobile devices using hardware tokens. In *USENIX Security*, pages 893–908. USENIX, 2014.

[18] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society, 2015.

[19] C-based Design. High-level synthesis with Vivado HLS. http://www.xilinx.com/products/design-tools/vivado.html, 2013.

[20] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. *Logic Synthesis*. McGraw-Hill, Inc., 1994.

[21] Encounter. RTL compiler by cadence design systems. http://www.cadence.com/products/di/pages/default.aspx.

[22] David Evans, Yan Huang, Jonathan Katz, and Lior Malka. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.

[23] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Cbmc-gc: An ansi c compiler for secure two-party computations. In *Compiler Construction*, pages 244–249. Springer, 2014.

[24] Tore K Frederiksen and Jesper B Nielsen. Fast and maliciously secure two-party computation using the GPU. In *ACNS*, pages 339–356. Springer, 2013.

[25] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, pages 405–422. Springer, 2014.

[26] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC*, pages 218–229. ACM, 1987.

[27] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 1996.

[28] 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 *CCS*. ACM, 2012.

[29] Mentor Graphics and Customer Success Story. HDL designer. http://www.mentor.com/products/fpga/hdl_design/hdl_designer_series/, 2008.

[30] Sumit Gupta, Rajesh Gupta, Nikil Dutt, and Alex Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer US, 2004.

[31] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: Tool for automating secure two-party computations. In *CCS*, pages 451–462. ACM, 2010.

[32] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *ASIACCS*, pages 437–446. ACM, 2013.

[33] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[34] Rolf Herken. *The universal Turing machine: a half-century survey*. Springer-Verlag New York, Inc., 1995.

[35] Yan Huang, Peter Chapman, and David Evans. Privacy-preserving applications on smartphones. In *USENIX HotSec*. USENIX, 2011.

[36] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, pages 539–554. USENIX, 2011.

[37] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *ACSAC*, pages 169–178. ACM, 2013.

[38] Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *CHES*, pages 383–397. Springer, 2010.

[39] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *S&P*, pages 216–230. IEEE, 2008.

[40] Gerry Kane and Joe Heinrich. *Mips risc architecture*, volume 1. Prentice Hall Englewood Cliffs, 1992.

[41] Jonathan Katz and Lior Malka. Constant-round private function evaluation with linear complexity. In *ASIACRYPT*, pages 556–571. Springer, 2011.

[42] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In *CRYPTO*, pages 440–457. Springer, 2014.

[43] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, pages 1–20. Springer, 2009.

[44] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, pages 486–498. Springer, 2008.

[45] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In *FC*, pages 83–97. Springer, 2008.

[46] Benjamin Kreuter, Abhi Shelat, Benjamin Mood, and Kevin RB Butler. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security*, pages 321–336. USENIX, 2013.

[47] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, pages 285–300. USENIX, 2012.

[48] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*. Springer, 2007.

[49] Yehuda Lindell and Benny Pinkas. A proof of Yao's protocol for secure two-party computation. *Journal of Cryptology*, pages 161–188, 2009.

[50] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 2012.

[51] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient RAM-model secure computation. In *S&P*, pages 623–638. IEEE, 2014.

[52] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In *EUROCRYPT*, pages 719–734. Springer, 2013.

[53] Lior Malka. Vmcrypt: modular software architecture for scalable secure computation. In *CCS*, pages 715–724. ACM, 2011.

[54] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay-secure two-party computation system. In *USENIX Security*, pages 287–302. USENIX, 2004.

[55] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[56] Alan Mishchenko et al. ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2007.

[57] Payman Mohassel and Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *EUROCRYPT*, pages 557–574. Springer, 2013.

[58] Moni Naor, Benny Pinkas, and Reuben Sumner. Privacy preserving auctions and mechanism design. In *EC*, pages 129–139. ACM, 1999.

[59] PandA. A framework for hardware-software co-design of embedded systems. http://panda.dei.polimi.it/.

[60] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *ASIACRYPT*, pages 250–267. Springer, 2009.

[61] Shi Pu and Jyh-Charn Liu. Computing privacy-preserving edit distance and Smith-Waterman problems on the GPU architecture. Cryptology ePrint Archive 2013/204, 2013.

[62] Michael O Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive 2005/187, 2005.

[63] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *S&P*, pages 655–670. IEEE, 2014.

[64] Steve Rhoads. Plasma-most MIPS I (tm) opcodes: Overview. *Internet: http://opencores. org/project, plasma [May 2, 2012]*, 2006.
[65] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for NC^1. In *FOCS*, pages 554–566. IEEE, 1999.
[66] Ellen M. Sentovich, Kanwar J. Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, EECS, UC Berkeley, 1992.
[67] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 1936.
[68] Leslie G Valiant. Universal circuits (preliminary report). In *STOC*, pages 196–203. ACM, 1976.
[69] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.
[70] Andrew C-C Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167. IEEE, 1986.
[71] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *S&P*, pages 493–507. IEEE, 2013.
[72] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Eurocrypt*, 2015. To appear. Preliminary version: http://eprint.iacr.org/2014/756.
[73] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*, pages 99–112. Springer, 2008.

APPENDIX A
OPEN SOURCE LOGIC SYNTHESIS TOOLS

TinyGarble offers a generic methodology for generating GC that is transparent to the underlying logic synthesis tool. To show this point, we demonstrate an implementation of TinyGarble using the Yosys [69] and ABC [56] logic synthesis tool chain for circuit generation. Both of these tools are open-source and available online. We compare the performance of the commercial HDL synthesis tool, i.e., Synopsys DC, with this open-source synthesis tool chain. ABC is an academic package developed at the University of California Berkeley. Yosys is an HDL-based synthesis tool which calls ABC for its technology mapping. The HDL inputs for describing both sequential and combinational circuits are written in the Verilog programming language.

We compare the performance of these open-source tools to the commercially available Synopsys DC. The results are presented in Table V. For comparison purposes, we compute GTD and MFE using the netlists generated by Synopsys DC as reference. For most of the benchmarks GTDs are either very small or zero which implies that the number of non-XOR gates in circuits generated by Yosys and by Synopsys DC are almost similar. In terms of memory footprint, different tools perform better for different benchmark functions. These results shows that TinyGarble is transparent to the underlying logic synthesis tool as long as the tool is up to date with respect to the known methods for logic minimization and mapping. Since the logic synthesis tools perform a series of optimizations, they may use different (heuristic) algorithms for some of their internal steps which could lead to slightly different results. A user can choose between different synthesis tools based on their performance and availability.

TABLE V: Comparison of circuit generation performance between the commercial Synopsys DC and Yosys+ABC open source logic synthesizer.

Function	c	Synopsys DC		Yosys		GTD	MFE
		Non -XOR	Total gates	Non -XOR	Total gates		
Compare 16284	1	16,384	65,536	16,383	32,767	0%	2.00
	16,384	1	4	1	3	0%	1.33
Hamming 160	1	158	1,039	158	1,158	0%	0.90
	32	10	41	10	48	0%	0.85
Hamming 1600	1	1,597	10,679	1,597	11,364	0%	0.94
	320	13	47	13	57	0%	0.82
Hamming 16000	1	15,994	107,226	15,994	112,421	0%	0.95
	3,200	16	53	16	66	0%	0.80
Sum 128	1	127	634	127	763	0%	0.83
	64	2	10	2	14	0%	0.71
	128	1	5	1	7	0%	0.71
Sum 256	1	255	1,274	255	1,531	0%	0.83
	256	1	5	1	7	0%	0.71
Sum 1024	1	1,023	5,114	1,023	6,139	0%	0.83
	1,024	1	5	1	7	0%	0.71