Garblet: Multi-party Computation for Protecting Chiplet-based Systems

Mohammad hashemi, Shahin Tajik and Fatameh Ganji

Worcester Polytechnic Institute, Worcester, USA {mhashemi,stajik,fganji}@wpi.edu

Abstract. The introduction of shared computation architectures assembled from heterogeneous chiplets introduces new security threats. Due to the shared logical and physical resources, an untrusted chiplet can act maliciously to surreptitiously probe the data communication between chiplets or sense the computation shared between them. This paper presents Garblet, the first framework to leverage the flexibility offered by chiplet technology and Garbled Circuits (GC)-based MPC to enable efficient, secure computation even in the presence of potentially compromised chiplets. Our approach integrates a customized hardware Oblivious Transfer (OT) module and an optimized evaluator engine into chiplet-based platforms. This configuration distributes the tasks of garbling and evaluating circuits across two chiplets, reducing communication costs and enhancing computation speed. We implement this framework on an AMD/Xilinx UltraScale+ multi-chip module and demonstrate its effectiveness using benchmark functions. Additionally, we introduce a novel circuit decomposition technique that allows for parallel processing across multiple chiplets to further improve computational efficiency. Our results highlight the potential of chiplet systems for accelerating GC (e.g., the time complexity of garbled AES is 0.0226ms) in order to guarantee the security and privacy of the computation on chiplets.

Keywords: Multiparty computation $\,\cdot\,$ Garbled circuits Oblivious transfer $\,\cdot\,$ Chiplet $\,\cdot\,$ Heterogeneous Integration

1 Introduction

There is a significant shift in the chip industry from large, monolithic chip fabrication to modular architectures built from heterogeneous chiplets. Chiplets are designed as a response to the need for high performance and maximum efficiency while attempting to manage costs associated with manufacturing and yield. These heterogeneous designs also offer advantages for combining chiplets, fabricated potentially with older technologies by a trusted facility, with ones built with cutting-edge technologies with no security features. In such cases, while the slower trusted chiplet can act as the root of trust, computationally heavy tasks can be distributed among the faster cutting-edge chiplets.

However, such multi-chip modules create some novel security risks. Since chiplets could be manufactured by various suppliers or programmed by different designers, they could act maliciously and threaten the security and privacy of computation running on them, see Fig 1. Even if all chiplets are trusted, the interposer connecting them together could act maliciously and intercept the communication between multiple chiplets.

Hence, we ask the following research question: In a highly adversarial environment, where virtually all chiplets or interposers can act maliciously, is it still possible to perform

This is the author version of the paper accepted for the presentation at the The IEEE VLSI Test Symposium (VTS) 2025.



Figure 1: Threats of malicious chiplets and interposers in multi-chip modules.

distributed computation securely? This is answered positively by secure multiparty computation (MPC), particularly Yao's Garbled Circuits (GC), which enables parties to jointly evaluate functions without exposing underlying secret data. GCs allow two parties (the garbler and the evaluator) to jointly compute a function over their private inputs while keeping these inputs hidden from each other [BHR12]. Compared to other approaches to secure computation, namely fully homomorphic encryption (FHE), GC incurs much lower computation complexity, although at the cost of communication complexity [BCM⁺19].

In fact, communication can be a bottleneck for traditional GC use cases on untrusted cloud servers. The communication cost involves (1) the cost of exchanging inputs between the users and (2) the cost of running the primitive responsible for obliviously sending these inputs [CPS14]. The communication cost can be especially significant when the circuit is large. In such scenarios, the communication bandwidth is a key determinant. In this regard, two implementation paradigms have been identified: (1) sequentially transmitting the garbled tables' inputs, and (2) transmitting the entire circuit to be evaluated without revealing anything but the output [BELO16]. In the case of classical server communications, the former matches communication characteristics over a local area network (LAN), while the latter can be conducted much faster over a wide-area network (WAN). However, the communication *inside a chip*, has not been discussed in the MPC-related literature. In fact, chiplet-based systems enable high-speed, low-latency communication between individual chiplets [Xila], making them ideal for enhancing the performance of secure computations, particularly GC.

In this regard, our paper's **contributions** are as follows. (1) For the first time, our paper demonstrates the feasibility of GC implementation on chiplets, enabling secure MPC even in the presence of distrustful/corrupted chiplets. Our framework, Garblet, showcases a great deal of overhead reduction compared to conventional server-client MPC. This is thanks to our chiplet-based implementation of oblivious communication as well as the flexibility offered by chiplet-based systems. In addition to minimizing communication overhead, Garblet addresses the scalability challenges commonly encountered in secure computation. Through a novel circuit decomposition technique, Garblet distributes computation tasks across multiple chiplets to enable parallel execution and, consequently, significantly reduces computation time. Another key advantage of Garblet is its ability to enhance security through hardware-level isolation. Security-critical tasks, such as encryption, can be physically separated from nonsensitive operations, limiting the potential impact of an attack on non-critical components.

2 Background and Adversary Model

2.1 Garbled Circuits

Yao's Garbled Circuits (GC) is a foundational method in two-party Secure Function Evaluation (SFE) for Boolean circuits [LP04, HHNZ19]. This technique involves a garbler and an evaluator, enabling them to jointly compute a function without revealing their private inputs. The following key components and optimizations are essential in the GC

$w_{b}^{a^{0/1}}$									
Input	Garbled Input	XOR Output	Garbled XOR Output	AND Output	Garbled AND Output				
0,0	w_{a}^{0}, w_{b}^{0}	0	$E_{w_{a}^{0},w_{b}^{0}}(w_{c}^{0})$	0	$E_{w_{a}^{0},w_{b}^{0}}(w_{c}^{0})$				
0,1	w_a^0, w_b^1	1	$E_{w^0_a,w^1_b}(w^1_c)$	0	$E_{w_{a}^{0},w_{b}^{1}}(w_{c}^{0})$				
1,0	w_{a}^{1}, w_{b}^{0}	1	$E_{w_{a}^{1},w_{b}^{0}}(w_{c}^{1})$	0	$E_{w_{a}^{1},w_{b}^{0}}(w_{c}^{0})$				
1,1	w_a^1, w_b^1	0	$E_{w_a^1,w_b^1}(w_c^0)$	1	$E_{w_{a}^{1},w_{b}^{1}}(w_{c}^{1})$				

Figure 2: Garbled gates look-up table cf. [HFG24].

framework:

Oblivious Transfer (OT). In a 1-out-of-2 OT, the sender holds two messages m_0 and m_1 , while the receiver selects one message using a bit $i \in \{0, 1\}$, learning only m_i without revealing i to the sender. This mechanism is crucial for input transfer in GC protocols, ensuring privacy [Kil88].

Garbling Process. The garbler, denoted as P_1 , generates a garbled circuit by assigning random secrets w_i^j representing wire values, where $j \in \{0, 1\}$ for each wire W_i . Each gate G_i is encrypted into a garbled table T_i so that the evaluator, P_2 , can evaluate the circuit using these tables without learning the underlying inputs [BHR12]. Fig. 2 illustrates a typical garbled gate setup for an AND and an XOR gate. Practical implementations of GC leverage fixed-key block ciphers, namely, AES (part of $E_{\cdot,\cdot}(\cdot)$ in Fig. 2), to efficiently garble non-XOR gates with a unique identifier for each gate. In practice, multiple optimization techniques have been proposed to efficiently instantiate GCs, including free-XOR, row-reduction, and half-gate; see [HK19] for details.

2.2 Adversary Model

We assume that chiplets are deployed to perform distributed computations. We assume that at least one chiplet in the systems is trusted and can initiate the distributed computation. The adversary attempts to gain access to untrusted chiplets to at least corrupt one chiplet. In doing so, she can attempt to eavesdrop on the communication between chiplets or extract the users' data being processed on an infected chiplet. As a countermeasure, one can rely on GC-based computation between chiplets, where the security against *passive* adversary is provably guaranteed.

3 Methodology

In this section, we first elaborate on the client-server model and its mapping to Chiplet then the implementation of circuit decomposition using the reverse logic tracing approach [Che83], which divides the given circuit into sub-circuits. After that, each subcircuit is assigned to dedicated engines for parallel processing. For securely and obliviously transferring inputs, we implement a novel dedicated hardware OT module. For GC generation, one can use any existing garbling engine, e.g., FASE [HK19]. We also create the very first evaluator engine on the chiplet to work efficiently with the garbling engine.

3.1 Client-Server Model and Chiplet Mapping

The design of Garblet leverages the traditional server-client model often used in secure computation frameworks. In this model, the garbler acts as the server and handles the majority of computationally intensive tasks, such as generating garbled tables and managing 4

```
Algorithm 1: Circuit Decomposition
   Input: Circuit f with outputs O and inputs I
   Output: Set of sub-circuits C, one for each output in O
1 Step 1: Extract Outputs and Inputs
2 Let O \leftarrow \texttt{ExtractOutputs}(f);
                                                                 // Extract output nodes
3 Let I \leftarrow \texttt{ExtractInputs}(f);
                                                                  // Extract input nodes
4 Step 2: Initialize Sub-Circuit Set
 5 Initialize an empty set C to store sub-circuits for each output.
6 Step 3: Reverse Logic Tracing
7 foreach output o \in O do
       Initialize sub-circuit C_o for o.
 8
       Call ReverseTraverse(o, f, C_o);
                                                                 // Trace back to inputs
9
       // Recursive tracing function
10
       ReverseTraverse(n, f, C_o) if n is an input node then
       return ;
                                                               // Stop at primary input
11
       Add n and its gate to C_o.
12
       foreach input i of gate n do
13
          ReverseTraverse(i, f, C_o);
                                                              // Recurse on gate inputs
\mathbf{14}
15 Step 4: Construct and Optimize Sub-Circuits
16 foreach o \in O do
       C_o \leftarrow \texttt{ConstructSubCircuit}(C_o);
17
                                                                 // Compile traced gates
      C_o \leftarrow \texttt{OptimizeCircuit}(C_o);
18
                                                                 // Minimize redundancy
19 Return C;
                                         // Return the set of optimized sub-circuits
```

cryptographic keys. The evaluator, acting as the client, performs less computationally demanding tasks, primarily focused on decrypting the garbled tables and evaluating the circuit. This division of roles ensures that the evaluator's operations are optimized for speed, enabling real-time application scenarios, while the garbler focuses on heavy computation with higher resource requirements.

In Garblet, this client-server model is mapped to a chiplet-based architecture, where one chiplet functions as the garbler chiplet and the other as the evaluator chiplet. The garbler chiplet is equipped with dedicated hardware modules, including AES-based encryption units, a key management unit, and a pipelined garbling engine to efficiently handle the resource-intensive garbling process. In contrast, the evaluator chiplet is designed for low-latency operations, integrating modules for secure OT protocol execution and optimized evaluator engines.

3.2 Reverse Logic Tracing for Circuit Decomposition

Reverse logic tracing decomposes a circuit by backtracking from each output node to its input dependencies, capturing all gates, wires, and connections involved in the computation. This process is illustrated in Algorithm 1. The decomposition starts by selecting the primary outputs and performing a depth-first traversal (DFT) [Che83] in reverse, tracing the logic back to the primary inputs (PIs). During this traversal, each gate and input affecting the output is marked, forming a complete dependency tree. After constructing these dependency trees, each sub-circuit is compiled as an isolated block with all necessary components. The final step involves optimizing each sub-circuit for logic redundancy by simplifying or combining gates and paths that do not directly impact the output, thereby reducing complexity while maintaining full functionality.

By dividing circuits into sub-circuits using reverse logic tracing, our methodology

facilitates parallel processing, critical for scaling to larger and more complex circuits. The number of sub-circuits generated increases linearly with the number of circuit outputs, enabling finer granularity in workload distribution. While this increases pre-processing time slightly, it significantly enhances scalability by allowing multiple garbling and evaluator engines to operate concurrently, as demonstrated in Section 4.1.

As an example, Fig. 3 shows the sub-circuits of a two-bit adder, with each dashed area representing a sub-circuit responsible for a specific output bit. The gray dashed area is the G1 gate connected to the S0 output and A0 and B0 inputs. The dark blue dashed area is the sub-circuit of G2, G3, and G4 gates connected to S1 output and all the inputs. The third sub-circuit includes G7, G6, G5, G3, and G2 connected to the C0 output and all the circuit inputs.

3.3 Oblivious Transfer Implementation

The OT module enables secure transmission of data between the Garbler and Evaluator chiplets. We implemented a hardware-based 1-out-of-2 OT module, which comprises three primary blocks: the Key Generator, Random Selector, and Communication Interface, each designed in Verilog and synthesized onto the chiplets.

The Key Generator Block on the Garbler Chiplet uses a True Random Number Generator (TRNG) and key management unit to produce cryptographic keys for each input wire. The TRNG outputs are processed by a Von Neumann extractor [Per92] to ensure uniform distribution. For each input wire W_i , a random key K_i^0 is generated, and $K_i^1 = K_i^0 \oplus \delta$, where δ is a secure global offset. These keys are stored in dual-port BRAM, with retrieval managed by a control unit.

The Random Selector Block on the Evaluator Chiplet generates a random bit s_i for each wire, determining which key $(K_i^0 \text{ or } K_i^1)$ will be used. Each selection bit s_i is masked with a one-time pad r_i to form $m_i = s_i \oplus r_i$, then stored in a FIFO buffer. The masked bits m_i are sent to the Garbler Chiplet via the Communication Interface using a handshaking protocol.

The Communication Interface supports secure data exchange using high-speed protocols such as AXI and PCIe. Data channels use AXI4-Lite for control and AXI4-Stream for high-bandwidth data transfer, with data encrypted before transmission to ensure privacy. Dual-port BRAM buffers incoming and outgoing data, with a control FSM managing data flow to prevent conflicts.

OT Execution. The Garbler Chiplet initializes its TRNG and generates a unique global offset δ while the Evaluator Chiplet pre-loads the random bits r_i . The Garbler Chiplet generates keys K_i^0 and K_i^1 for each input wire and stores them. The evaluation chiplet generates selection bits s_i , masks them with r_i to produce $m_i = s_i \oplus r_i$, and sends them to the Garbler chiplet. The Garbler Chiplet computes the selected key $K_i^{s_i} = K_i^{m_i \oplus r_i}$ and transmits it to the Evaluator Chiplet, which stores the keys for evaluation.

3.4 Evaluator Engine Implementation

The literature suggested that a garbling engine can be modified to create an evaluator engine [HK19]; however, based on our experience, this task is more delicate than expected. For implementing the evaluator engine, one needs to implement input key handling, decryption logic, and memory management, as detailed in Algorithm 2. In our efficient and practical evaluator engine, only a single key K_i per input wire is needed. The hardware OT module transmits the evaluator's selected keys, which are stored in dual-port BRAM. In this way, for each wire W_i , BRAM_{Key} $[i] = K_i$. A synchronization unit ensures that evaluation starts only after the secure storage of keys. Here, secure storage means that all keys are stored in the isolated memory and all buffers are free, i.e., the handshake signal and acknowledge signal are both raised to 1; otherwise, the key values are accessible during

A	Algorithm 2: Evaluator Engine Implementation						
	Input: Garbled Circuit C, Garbled Tables T, Input Keys K_{in} Output: Final Output Keys K_{out}						
1 2 3	InitializeMMU(); Initialize dual-port BRAM for input keys and garbled tables; Configure AES cores for decryption mode;						
4 5 6	ReceiveKeys(K_{in}); Receive the evaluator's input keys through the OT protocol and store them in BRAM; Synchronize with Garbler Chiplet to ensure all keys are securely stored;						
7 8 9	$ \begin{array}{c c} \textbf{for each gate } G_i \ in \ C \ \textbf{do} \\ & &$						
10 11	else $\begin{tabular}{ c c c } \hline & \mathbf{k}_{\mathrm{out}} \leftarrow \mathtt{DecryptGate}(K_{\mathrm{in}},T_i) \ ; & \textit{// Decrypt using AES in decryption mode} \end{tabular}$						
12 13	Store K_{out} in BRAM for subsequent gate evaluations; ManageMemory();						
14	ManageMemory();						
15 16 17	Collect all output keys K_{out} corresponding to the circuit's primary outputs; Concatenate the keys to form the final output; TransmitData((<i>Final Output</i>));						
18	Send the final output to the external evaluator for verification or further use;						

the operations. The AES cores are reconfigured for decryption mode. For each gate G_i , the evaluator decrypts the garbled truth table entry E_i :

$$K_{\text{out}} = \text{AES}^{-1}(K_{\text{in}}, E_i) = \text{AES}^{-1}(K_{\text{in}}, \text{AES}(K_{\text{in}}, K_{\text{out}} \oplus R)),$$

where R is a random value. For XOR gates, the output key is directly computed: $K_{\text{out}} = K_{\text{in1}} \oplus K_{\text{in2}}$. This avoids decryption and reduces computational overhead. Memory management was optimized using a Memory Management Unit (MMU) that dynamically allocates addresses for sub-circuits being evaluated:

 $\mathrm{MMU}_{\mathrm{Address}}[i] = \left\{ \begin{array}{ll} \mathrm{read}; & \mathrm{if \ gate} \ G_i \ \mathrm{is \ ready} \ \mathrm{for \ evaluation}, \\ \mathrm{write}; & \mathrm{if \ output} \ K_{\mathrm{out}} \ \mathrm{is \ to \ be \ stored}. \end{array} \right.$

The MMU minimizes data collisions and ensures efficient memory access.

We also optimized the communication interface between the Garbler and Evaluator engines. The AXI4-Stream interface was configured for direct memory access (DMA), supporting bulk data transfers of garbled tables and keys. Each packet is encrypted and includes a parity check P and 32-bit cyclic redundancy check (CRC) to verify data integrity: $P = \bigoplus_{i=1}^{n} \text{bit}_{i}$.

3.5 Sub-circuit Assignment: Advantages and Process

Efficient sub-circuit assignment and synchronization of the garbling and evaluation phases are essential for optimizing performance in our framework. The scheduler dynamically allocates encryption keys to the garbling engines and meticulously tracks the progress of each sub-circuit to prevent conflicts and ensure smooth parallel execution. After the garbling phase, the generated garbled tables are transmitted to the evaluator engines, where they are processed, and the results are evaluated to form the final output. Fig. 4 illustrates the high-level flow of sub-circuit assignment and its integration into the overall



Figure 3: The sub-circuits of a two-bit adder corresponding to each output.

system. By distributing garbling tasks across multiple engines, the scheduler not only enhances computational efficiency but also contributes to the framework's security.

A crucial security benefit of this structured approach lies in Garblet's ability to enforce hardware-level isolation for security-critical tasks. By assigning different tasks, such as encryption and evaluation, to separate chiplets, we can physically separate sensitive operations (e.g., cryptographic key management) from non-critical ones (e.g., intermediate data storage and transmission). This separation limits the attack surface and reduces the risk of adversaries compromising critical components. For example, if an attacker gains access to a chiplet responsible for handling non-sensitive operations, such as managing communication between sub-circuits, they cannot directly manipulate or observe the encryption keys managed by a separate, isolated chiplet dedicated to a garbling engine. This hardware-level partitioning ensures that even if one component is compromised, the security of the overall computation remains intact, as sensitive operations are shielded from potential attacks. This robust isolation, combined with efficient sub-circuit assignment and synchronization, enables Garblet to perform secure computations with minimal performance overhead, providing both security and efficiency in a highly modular and scalable manner.

3.6 Chiplet-based GC Implementation Flow

To leverage the performance benefits of a chiplet-based implementation, we integrate a garbling engine, our custom hardware OT module, and our evaluator engine on chiplets. This integration enables us to perform complete GC without relying on external parties as opposed to, e.g., HostCPU in TinyGarble [SHS⁺15]. Fig. 5 illustrates the GC protocol distribution across two chiplets to reduce communication overhead and enhance secure computation efficiency. We utilize Xilinx UltraScale+ chiplets [Xile] that offer modular platforms with high-speed interfaces such as AXI [Xila] and PCIe [Xild]. The framework is implemented using Xilinx Vivado Design Suite [Xilf] and Vitis Unified Software Platform [Xilb]. These tools optimize communication latency between the garbler and evaluator chiplets.

The garbling process is assigned to Chiplet A, while Chiplet B performs the evaluation. This separation allows parallel operation with minimal delays. Chiplet A garbles each Boolean gate in four main phases: (I) circuit representation and key generation, (II) gate garbling, (III) pipelined garbling, and (IV) inter-chiplet communication.

Circuit Representation and Key Generation. The function is represented as a Boolean circuit where each gate (AND, OR, XOR) corresponds to a logical operation in GC. The Key Generator module produces two cryptographic keys per gate's wire, one for '0' and one for '1'; see Fig. 2.

Pipelined Garbling Process. Each gate is garbled in Chiplet A's garbling engine by



Figure 4: Sub-circuit assignment to garbling/evaluator engines.

creating a garbled truth table, where input wire keys are encrypted. The garbling engine is pipelined to garble one gate per clock cycle, ensuring continuous AES core operation. As gates are processed, garbled tables, inputs, and output keys are stored in Chiplet A's dual-port BRAM. A memory management wrapper manages read/write operations to prevent conflicts, enabling simultaneous garbling and data transmission to Chiplet B.

Inter-chiplet Communication. Communication between Chiplet A and Chiplet B is established using AXI and PCIe protocols to handle large data transfers efficiently. AXI enables direct memory access (DMA) for fast data transmission, while PCIe supports high-bandwidth communication to reduce delays in garbled table transfer. Chiplet B, configured as the evaluator, uses our HW 1-out-of-2 OT protocol to securely select its input keys. Upon receiving the garbled tables and keys, Chiplet B evaluates each gate using the garbled tables. XOR gates are evaluated without encryption due to Free-XOR optimization, while non-XOR gates are decrypted to reveal the correct output keys.

Evaluation and Synchronization. Dual-port BRAM in Chiplet B manages garbled tables and evaluation keys. Synchronization between Chiplets A and B is handled via a handshake protocol, ensuring Chiplet B only begins evaluation after receiving all required data. The Universal Chiplet Interface Express (UCIe) protocol optimizes synchronization, reducing delays and improving efficiency [Xild]. Once the evaluation is complete, Chiplet B decrypts the final garbled output to obtain the output in plaintext.

System Optimization. Parameters such as clock speed and inter-chiplet bandwidth are chosen for scalability to handle large computation efficiently [Xilc, Xila, Xile] (see Section 4.1).

This makes the framework suitable for real-time applications where quick and secure computations are crucial.



Figure 5: The flow of GC implementation on the chiplet-based system.

Table 1: Hardware resource utilization: comparison between Garblet and implementations on monolithic FPGAs [HK19, HRFG22].

Resource	Garbling E	ngine	Evaluator Engine					
itesource	FASE [UK10]	Carblet	Monolithic	Monolithic	Carblet			
	FASE [IIK19]	Garbiet	(Resource Efficient)	(High Performance)	Garbiet			
LUT	31330	11729	1775	94701	5717			
FF	11416	4103	1278	52534	2739			
LUTRAM	553	93	N/R	N/R	78			
BRAM	68.5	103	0	0	95			
DSP	0	1	0	0	1			

4 Results

4.1 Experimental Setup

We evaluated Garblet using common benchmark functions such as AES, multiplication, multiply-accumulate (MAC), and an 8-bit adder cf. [HK19]. The experiments were conducted in two distinct scenarios to compare performance and hardware utilization. In the first scenario, a traditional server-client (PC-FPGA) setup was used, where a PC with an Intel Core i7-7700 CPU @ 3.60GHz, 16 GB RAM, and Linux Ubuntu 20 acted as the garbler, and the evaluator was implemented on an ARTIX7 FPGA board operating at a clock frequency of 20 MHz. This configuration served as the baseline for performance comparison. In the second scenario, Garblet was implemented on Vertex UltraScale+ chiplets, where the garbler and evaluator were deployed on separate chiplets. These chiplets were configured to achieve the maximum possible frequency and bandwidth, with high-speed transceivers operating at 32.75Gb/s and clock frequencies reaching up to 600 MHz.

To ensure consistency and minimize variability caused by external factors, all execution times reported represent the average of five independent runs. We explored the impact of resource allocation by testing configurations with a single pair of garbling and evaluator engines and then scaling up to multiple engines. This analysis allowed us to evaluate the trade-offs between hardware costs and performance gains in both resource-efficient and high-performance modes.

Deserves	Garbling	Schodulor	Key	OT	Evaluator	Combiner	Controller	
nesource	Engine	Scheduler	Generator	Module	Engine	Compiler		
LUT	11729	5739	8088	4182	5717	47	2071	
FF	4103	3693	4270	2237	2739	13	1629	
LUTRAM	93	0	0	21	78	0	0	
BRAM	103	109	57	2	95	5	15	
DSP	1	0	0	0	1	0	0	

Table 2: Hardware resource utilization of Garblet individual modules.

Table 3: Execution time cost (in μ s): comparison of common benchmarks using baseline (not garbled), monolithic, and Garblet implementation.

Bonchmark	Gar	Garbling Time (μs)		Evaluation Time (μs)		Communication (µs)			Total (μ s)			
Dencimiark	Baseline	Monolithic	Garblet	Baseline	Monolithic	Garblet	Baseline	Monolithic	Garblet	Baseline	Monolithic	Garblet
Add_8_1	N/A	9.72	0.0173	N/A	0.619	0.00312	N/A	173,000	0.293	0.017	1,730	0.313
Mult_1024_204	8 N/A	3,910,000	30.1	N/A	4,180	9.21	N/A	8,950,000,000	4,270	257.17	8,960,000,000	4,300
MAC_32_1	N/A	828	9.31	N/A	99.1	3.02	N/A	576,000	11.0	43.85	577,000	842
AES_128_1	N/A	7,120	6.02	N/A	599	1.02	N/A	4,910,000	219	11.83	4,910,000	226

4.2 Hardware Resource Utilization Analysis

Table 1 compares the hardware resource utilization of Garblet with the implementations on monolithic FPGAs as in [HK19, HRFG22]. Below is a concise analysis highlighting the resource savings achieved by the Garblet.

The Garblet's garbling engine reduces LUT utilization by $2.67 \times$ (from 31,330 to 11,729 LUTs) compared to FASE [HK19] due to its modular design and DSP offloading. The evaluator's LUT usage shows a $16.57 \times$ improvement over the implementation on a monolithic FPGA, demonstrating significant efficiency gains. The Garblet garbling engine also uses 4,103 FFs compared to 11,416 FFs in FASE, achieving a $2.78 \times$ reduction, primarily due to the efficient use of dual-port BRAMs, which minimizes the dependency on flip-flops for intermediate storage. For LUTRAM, Garblet utilizes only 93 LUTRAMs compared to 553 in FASE, representing a $5.95 \times$ reduction attributed to the use of BRAMs for memory storage. The BRAM usage in the garbling engine increased by $1.5 \times$ (from 68.5 to 103 BRAMs), which is justified by the adoption of dual-port BRAMs for efficient data handling between garbling and evaluation engines. Each Garblet's garbling and evaluator engine incorporates 1 DSP, offloading specific computational tasks and further reducing LUT utilization. This trade-off is also justified by the significant reductions in computation time and enhanced scalability, which are critical for large-scale secure computations. Table 2 shows the resource utilization of each module. The garbling and evaluator engines can be instantiated multiple times as long as the platform supports the resource requirements.

4.3 Execution Time Cost

Effect of Communication Reduction.

We compared the execution time cost of common benchmark functions for implementations on baseline (not garbled), monolithic FPGAs, and Garblet. The Garblet framework was tested in two setups: one with a single garbling and evaluator engine for resource efficiency and another with multiple engines for better performance. Table 3 shows the execution times for benchmarks like AES, multiplication, MAC, and an 8-bit adder.

Garblet reduces communication costs by up to $59,000 \times$ compared to monolithic implementations and improves performance by up to $5,500 \times$ for benchmarks like the 8-bit adder. When compared to Baseline implementations, which do not involve multiple parties and therefore do not require garbling, evaluation, or communication, Garblet introduces additional overhead for security. For the benchmarks, Garblet is about 15.65 times slower for the 8-bit adder, 257 times slower for multiplication, 745 times slower for MAC, and 47.8 times slower for AES.

Table 4: Execution time comparison (in μ s) between monolithic and Garblet implementation with one and three engines.

Bonchmark	# Sub circuite	Garbling Time (µs)			Evaluation Time (μs)			Communication (µs)		
Dencimark	# Sub-circuits	Monolithia Garblet Ga		Garblet	Monolithio	Garblet	Garblet	Monolithio	Garblet	Garblet
		Monontine	One Engine	Three Engines	Monontine	One Engine	Three Engines	Monontine	One Engine	Three Engines
2-bit Adder	3	14	0.0591	0.0377	1.47	0.00628	0.00412	473,000	0.522	0.849
Mult_1024_2048	2048	3,910,000	30.1	12.9	4,180	9.21	4.66	8,950,000,000	4,270	6,190

Table 5: Execution time and peak memory cost of the circuit decomposition algorithm.

Benchmark	# Sub-circuits	Time (s)	Memory Peak (MB		
2-bit Adder	3	3.1	394		
Mult_1024_2048	2048	65901	11387		

Circuit Decomposition Execution Time: We evaluated the cost of our circuit decomposition algorithm on two benchmark functions: (I) a 2-bit adder and (II) Mult_1024_2048. The decomposition algorithm was implemented on the PC. Note that this pre-processing is performed offline (before running the GC protocol) and does not impact the framework's online performance. Multiple sub-circuits enable parallel computation, enhancing the framework's performance. Table 5 shows the execution time and peak memory cost of the circuit decomposition algorithm. As the number of sub-circuits increases, the execution time rises exponentially.

4.4 Acceleration Using Multiple Garbling/Evaluator Engines

The Garblet benefits extend beyond communication cost reduction. Using multiple garbling and evaluator engines, along with circuit decomposition, enables parallel execution of computation tasks, significantly improving overall performance. We evaluated this performance gain by running the 2-bit adder and Mult_1024_2048 benchmark functions using one and three garbling/evaluator engines. Table 4 shows the execution time cost comparison between monolithic Garblet with one and three engines. Using three engines significantly reduces garbling and evaluation times for both benchmark functions. For the 2-bit adder, garbling time decreased by $1.57\times$, and evaluation time was reduced by $1.52\times$. Communication time also shows a reduction, demonstrating minimal overhead with multiple engines. For the Mult_1024_2048 benchmark, the benefits of parallel processing are even more pronounced, with garbling time reduced by $2.34\times$ and evaluation time by $1.98\times$. These results demonstrate that using multiple engines significantly accelerates framework performance while minimizing communication costs. Parallel processing of sub-circuits enables efficient handling of complex computations, making it a viable solution for time-sensitive applications.

5 Conclusion

In this paper, we presented Garblet, a chiplet-based secure MPC framework that integrates custom hardware modules for OT and optimized evaluator engines. Through a novel circuit decomposition, Garblet distributes garbling and evaluation tasks across multiple chiplets, significantly reducing communication overhead and improving computational efficiency. Experimental results demonstrated that Garblet achieves up to a $59,000 \times$ reduction in communication costs and up to a $5,500 \times$ performance improvement in key benchmarks compared to traditional server-client setups. Overall, Garblet paves the way for future research and development in chiplet-based secure computation, providing a scalable and efficient solution to the growing demand for secure and efficient computation on chiplets.

References

- [BCM⁺19] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. Garbled neural networks are practical. *Cryptology ePrint Archive*, 2019.
- [BELO16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In Proc. of the Conference on Computer and Communications Security, pages 578–590, 2016.
- [BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Proc. of the 2012 ACM Conf. on Computer and Comm. security, pages 784–796, 2012.
- [Che83] To-Yat Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, (4):504–512, 1983.
- [CPS14] Ashish Choudhury, Arpita Patra, and Nigel P Smart. Reducing the overhead of mpc over a large population. In *International Conference on Security and Cryptography* for Networks, pages 197–217. Springer, 2014.
- [HFG24] Mohammad Hashemi, Domenic Forte, and Fatemeh Ganji. Time is money, friend! timing side-channel attack against garbled circuit constructions. In International Conference on Applied Cryptography and Network Security, pages 325–354. Springer, 2024.
- [HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General purpose compilers for secure multi-party computation. In 2019 IEEE symposium on security and privacy (SP), pages 1220–1237. IEEE, 2019.
- [HK19] Siam U Hussain and Farinaz Koushanfar. Fase: Fpga acceleration of secure function evaluation. In Intrl. Symp. on Field-Programmable Custom Computing Machines, pages 280–288. IEEE, 2019.
- [HRFG22] Mohammad Hashemi, Steffi Roy, Domenic Forte, and Fatemeh Ganji. Hwgn 2: Sidechannel protected nns through secure and private function evaluation. In Proc. of Security, Privacy, and Applied Cryptography Engineering, pages 225–248, 2022.
- [Kil88] Joe Kilian. Founding crytpography on oblivious transfer. In Proc. of the annual ACM Symp. on Theory of computing, pages 20–31, 1988.
- [LP04] Y Lindell and B Pinkas. A proof of yao's protocol for secure two-party computation. eccc report tr04-063. In *Electronic Colloquium on Computational Complexity (ECCC)*, 2004.
- [Per92] Yuval Peres. Iterating von neumann's procedure for extracting random bits. *The* Annals of Statistics, pages 590–597, 1992.
- [SHS⁺15] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In 2015 IEEE Symp. on Security and Privacy, pages 411–428. IEEE, 2015.
- [Xila] Xilinx. Exploring the PS-PL AXI interfaces on Zynq UltraScale+ MPSoC. [Online]https://j-marjanovic.io/exploring-ps-pl-axi-ultrascale[Accessed: Sep.22,2024].
- [Xilb] Xilinx. Vivits2022.1. [Online]https://docs.xilinx.com/v/u/en-US/ ug1416-vitis-documentation.html[AccessedSep.22,2024].
- [Xilc] Xilinx. Zynq UltraScale+ MPSoC APIs and AXI Communication. [Online]https:// www.amd.com/en/products/soc/zynq-ultrascale-plus[Accessed:Sep.22,2024].
- [Xild] Xilinx. Zynq UltraScale+ MPSoC Documentation. [Online]https://www.xilinx. com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html[Accessed: Sep.22,2024].
- [Xile] Xilinx. Zynq UltraScale+ MPSoC PS-PL Interface. [Online]https://xilinx-wiki. atlassian.net/wiki/spaces/A/pages/444006775/Zynq[Accessed:Sep.22,2024].
- [Xilf] Xilinx, Inc. Vivado2022.1. [Online]https://www.xilinx.com/products/ design-tools/vivado.html[AccessedSep.22,2024].