4

Implementation Techniques

Although secure computation protocols (as described in Chapter 3) were known since the 1980s, the first full implementation of a generic secure computation system was Fairplay (Malkhi *et al.*, 2004). Fairplay compiles a high-level description of a function into a circuit, described using a custom-designed Secure Hardware Description Language (SHDL). This circuit could then be executed as a protocol by a generator program and an evaluator program running over a network.

As a rough indication of the costs of MPC with Fairplay, the largest benchmark reported for Fairplay was finding the median of two sorted input arrays containing ten 16-bit numbers from each party. This required executing 4383 gates, and took over 7 seconds on a local area network (the time was then dominated by the oblivious transfer, not the garbled circuit execution). Modern MPC frameworks can execute millions of gates per second, and scale to circuits computing complex functions on large inputs, with hundreds of billions of gates.

Perhaps a factor of ten of the improvement can be attributed to general improvements in computing and network bandwidth (the Fairplay results are on a LAN with 618 Mbps, compared to 4 Gbps routinely available today), but the rest of the 3–4 orders of magnitude improvements are due primarily to the

advances in implementation techniques described in this chapter. These include optimizations that reduce the bandwidth and computational costs of executing a GC protocol (Section 4.1), improved circuit generation (Section 4.2), and protocol-level optimizations (Section 4.3). We focus on improvements to Yao's GC protocol, as the most popular generic MPC protocol, although some of the improvements discussed apply to other protocols as well. Section 4.4 briefly surveys tools and languages that have been developed for implementing privacy-preserving applications using MPC.

4.1 Less Expensive Garbling

The main costs of executing a garbled circuits protocol are the bandwidth required to transmit the garbled gates and the computation required to generate and evaluate the garbled tables. In a typical setting (LAN or WAN and moderate computing resources such as smartphone or a laptop), bandwidth is the main cost of executing GC protocols. There have been many improvements to the traditional garbling method introduced in Section 3.1.2; we survey the most significant ones next. Table 4.1 summarizes the impact of garbing improvements on the bandwidth and computation required to generate and evaluate a garbled gate. We described point-and-permute in Section 3.1.1; the other techniques are described in the next subsections.

Technique	size		calls to H	
	XOR	AND	XOR	AND
classical	4	4	4	4
point-and-permute (1990) (§3.1.1)	4	4	4, 1	4, 1
row reduction (GRR3) (1999) (§4.1.1)	3	3	4, 1	4, 1
FreeXOR + GRR3 (2008) (§4.1.2)	0	3	0	4, 1
half gates (2015) (§4.1.3)	0	2	0	4, 2

Table 4.1: Garbling techniques (based on Zahur *et al.* (2015)). Size is number of "ciphertexts" (multiples of κ bits) transmitted per gate. Calls to H is the number of evaluations of H needed to evaluate each gate. When the number is different for the generator and evaluator, the numbers shown are the *generator calls*, *evaluator calls*.

4.1.1 Garbled Row Reduction

Naor *et al.* (1999) introduced *garbled row reduction* (GRR) as a way to reduce the number of ciphertexts transmitted per gate. The key insight is that it is not necessary for each ciphertext to be an (unpredictable) encryption of a wire label. Indeed, one of the entries in each garbled table can be fixed to a predetermined value (say 0^κ), and hence need not be transmitted at all. For example, consider the garbled table below, where a and b are the input wires, and c is the output:

$H(a^1 \,\|\, b^0) \oplus\ c^0$
$H(a^0 \,\|\, b^0) \oplus\ c^0$
$H(a^1 \,\|\, b^1) \oplus\ c^1$
$H(a^0 \,\|\, b^1) \oplus\ c^0$

Since c^0 and c^1 are just arbitrary wire labels, we can select $c^0 = H(a^1 \,\|\, b^0)$. Thus, one of the four ciphertexts in each gate (say, the first one when it is sorted according point-and-permute order) will always be the all-zeroes string and does not need to be sent. We call this method *GRR3* since only three ciphertexts need to be transmitted for each gate.

Pinkas *et al.* (2009) describe a way to further reduce each gate to two ciphertexts, applying a polynomial interpolation at each gate. Because this is not compatible with the FreeXOR technique described next, however, it was rarely used in practice. The later half-gates technique (Section 4.1.3) achieves two-ciphertext AND gates and is compatible with FreeXOR, so supersedes the interpolation technique of Pinkas *et al.* (2009).

4.1.2 FreeXOR

One of the results of Kolesnikov (2005) was the observation that the GESS sharing for XOR gates can be done without any growth of the share sizes (Section 3.6). Kolesnikov (2005) found a lower bound for the minimum share sizes, explaining the necessity of the exponential growth for independent secrets. This bound, however, did not apply to XOR gates (or, more generally, to "even" gates whose truth table had two zeros and two ones).

As introduced in Section 3.6, XOR sharing for GESS can simply be done as follows. Let $s_0, s_1 \in \mathcal{D}_\mathcal{S}$ be the output wire secrets. Choose $R \in_R \mathcal{D}_\mathcal{S}$ and

set the shares $sh_{10} = R, sh_{11} = s_0 \oplus s_1 \oplus R, sh_{20} = s_0 \oplus R, sh_{21} = s_1 \oplus R$. The share reconstruction procedure on input sh_{1i}, sh_{2j}, outputs $sh_{1i} \oplus sh_{2j}$. It is easy to verify that this allows the evaluator to reconstruct the correct gate output secret. Indeed, e.g., $sh_{11} \oplus sh_{21} = (s_0 \oplus s_1 \oplus R) \oplus (s_1 \oplus R) = s_0$.

Denoting $s_0 \oplus s_1 = \Delta$, we observe that this offset Δ is preserved for the labels of each wire: $sh_{10} \oplus sh_{11} = sh_{20} \oplus sh_{21} = s_0 \oplus s_1 = \Delta$.

Because the shares on all of the gate's wires have the same offset, the above GESS XOR gate construction cannot be directly plugged into Yao's GC, since standard Yao's GC requires wires to be independently random. This breaks both correctness and security of GC. Indeed, correctness is broken because GESS XOR will not produce the pre-generated output wire secrets. Standard GC security proofs fail since GESS XOR creates correlations across wire labels, which are encryption keys. Even more problematic with respect to security is the fact that keys and encrypted messages are related to each other, creating *circular* dependencies.

FreeXOR: Integrating GESS XOR into GC. FreeXOR is a GC technique introduced by Kolesnikov and Schneider (2008b). Their work is motivated by the fact that in GESS an XOR gate costs nothing (no garbled table needed, and share secrets don't grow), while traditional Yao's GC pays full price of generating and evaluating a garbled table for XOR gates. The GC FreeXOR construction enables the use of GESS XOR construction by adjusting GC secrets generation to repair the correctness broken by the naïve introduction of GESS XOR into GC, and observing that a strengthening of the assumptions on the encryption primitives used in the construction of GC garbled tables is sufficient for security of the new scheme.

FreeXOR integrates GESS XOR into GC by requiring that *all* the circuit's wires labels are generated with the same offset Δ. That is, we require that for each wire w_i of GC $\widehat{C}$ and its labels w_i^0, w_i^1, it holds that $w_i^0 \oplus w_i^1 = \Delta$, for a randomly chosen $\Delta \in_R \{0,1\}^\kappa$. Introducing this label correlation enables GESS XOR to correctly reconstruct output labels.

To address the security guarantee, FreeXOR uses Random Oracle (RO) for encryption of gates' output labels, instead of the weaker (PRG-based) encryption schemes allowed by Yao GC. This is necessary since inputs to different instances of H are correlated by Δ, and furthermore different values

masked by H's output are also correlated by Δ. The standard security definition of a PRG does not guarantee that the outputs of H are pseudorandom in this case, but a random oracle does. Kolesnikov and Schneider mention that a variant of *correlation robustness*, a notion weaker than RO, is sufficient (Kolesnikov and Schneider, 2008b). In an important theoretical clarification of the FreeXOR required assumptions, Choi *et al.* (2012b) show that the standard notion of correlation robustness is indeed not sufficient, and pin down the specific variants of correlation robustness needed to prove the security of FreeXOR.

The full garbling protocol for FreeXOR is given in Figure 4.1. The FreeXOR GC protocol proceeds identically to the standard Yao GC protocols of Figure 3.2, except that in Step 4, P_2 processes XOR gates without needing any ciphertexts or encryption: for an XOR-gate G_i with garbled input labels $w_a = (k_a, p_a), w_b = (k_b, p_b)$, the output label is directly computed as $(k_a \oplus k_b, p_a \oplus p_b)$.

Kolesnikov *et al.* (2014) proposed a generalization of FreeXOR called fleXOR. In fleXOR, an XOR gate can be garbled using 0, 1, or 2 ciphertexts, depending on structural and combinatorial properties of the circuit. FleXOR can be made compatible with GRR2 applied to AND gates, and thus supports two-ciphertext AND gates. The half gates technique described in the next section, however, avoids the complexity of fleXOR, and reduces the cost of AND gates to two ciphertexts with full compatibility with FreeXOR.

4.1.3 Half Gates

Zahur *et al.* (2015) introduced an efficient garbling technique that requires only two ciphertexts per AND gate and fully supports FreeXOR. The key idea is to represent an AND gate as XOR of two *half gates*, which are AND gates where one of the inputs is known to one of the parties. Since a half gate requires a garbled table with two entries, it can be transmitted using the garbled row reduction (GRR3) technique with a single ciphertext. Implementing an AND gate using half gates requires constructing a *generator half gate* (where the generator knows one of the inputs) and an *evaluator half gate* (where the evaluator knows one of the inputs). We describe each half gate construction next, and then show how they can be combined to implement an AND gate.

Generator Half Gate. First, consider the case of an AND gate where the input wires are a and b and the output wire is c. The generator half-AND gate

PARAMETERS:
Boolean Circuit C implementing function $\mathcal{F}$, security parameter κ.
Let $H : \{0,1\}^* \mapsto \{0,1\}^{\kappa+1}$ be a hash function modeled by a RO.

PROTOCOL:

1. Randomly choose global key offset $\Delta \in_R \{0,1\}^\kappa$.
2. For each input wire w_i of C, randomly choose its 0 label,
$$w_i^0 = (k_i^0, p_i^0) \in_R \{0,1\}^{\kappa+1}.$$
Set the other label $w_i^1 = (k_i^1, p_i^1) = (k_i^0 \oplus \Delta, p_i^0 \oplus 1)$.
3. For each gate G_i of C in topological order
   (a) If G_i is an XOR-gate $w_c = \mathsf{XOR}(w_a, w_b)$ with input labels $w_a^0 = (k_a^0, p_a^0), w_b^0 = (k_b^0, p_b^0), w_a^1 = (k_a^1, p_a^1), w_b^1 = (k_b^1, p_b^1)$:
      i. Set garbled output value $w_c^0 = (k_a^0 \oplus k_b^0, p_a^0 \oplus p_b^0)$
      ii. Set garbled output value $w_c^1 = (k_a^0 \oplus k_b^0 \oplus \Delta, p_a^0 \oplus p_b^0 \oplus 1)$.
   (b) If G_i is a 2-input gate $w_c = g_i(w_a, w_b)$ with garbled labels $w_a^0 = (k_a^0, p_a^0), w_b^0 = (k_b^0, p_b^0), w_a^1 = (k_a^1, p_a^1), w_b^1 = (k_b^1, p_b^1)$:
      i. Randomly choose output label $w_c^0 = (k_c^0, p_c^0) \in_R \{0,1\}^{\kappa+1}$
      ii. Set output label $w_c^1 = (k_c^1, p_c^1) = (k_c^0 \oplus \Delta, p_c^0 \oplus 1)$.
      iii. Create G_i's garbled table. For each of 2^2 possible combinations of G_i's input values $v_a, v_b \in \{0,1\}$, set
$$e_{v_a,v_b} = H(k_a^{v_a} || k_b^{v_b} || i) \oplus w_c^{g_i(v_a,v_b)}.$$
      Sort entries e in the table by the input pointers, placing entry e_{v_a,v_b} in position $\langle p_a^{v_a}, p_b^{v_b} \rangle$.
4. Compute the output tables, as in Figure 3.2.

Figure 4.1: FreeXOR Garbling

computes $v_c = v_a \wedge v_b$, where v_a is somehow known to the circuit generator. Then, when v_a is false, the generator knows v_c is false regardless of v_b; when v_a is true, $v_c = v_b$. We use a^0, b^0, and c^0 to denote the wire labels encoding false for wires a, b, and c respectively. Using the FreeXOR design, the wire label for b is either b^0 or $b^1 = b^0 \oplus \Delta$. The generator produces the two ciphertexts:

$$H(b^0) \oplus c^0$$
$$H(b^1) \oplus c^0 \oplus v_a \cdot \Delta$$

These are permuted according to the pointer bits of b^0, according to the point-and-permute optimization.

To evaluate the half gate and obtain $v_a \wedge v_b$, the evaluator takes a hash of its wire label for b (either b^0 or b^1) and decrypts the appropriate ciphertext. If the evaluator has b^0, it can compute $H(b^0)$ and obtain c^0 (the correct semantic false output) by xor-ing it with the first ciphertext. If the evaluator has $b^1 = b^0 \oplus \Delta$, it computes $H(b^1)$ to obtain $c^0 \oplus v_a \cdot \Delta$. If $v_a = 0$, this is c^0; if $v_a = 1$, this is $c^1 = c^0 \oplus \Delta$. Intuitively, the evaluator will never know both b^0 and b^1, hence the inactive ciphertext appears completely random. This idea was also used implicitly by Kolesnikov and Schneider, 2008b, Fig. 2, in the context of programming components of a universal circuit.

Crucially for performance, the two ciphertexts can be reduced to a single ciphertext by selecting c^0 according to the garbled row-reduction (Section 4.1.1).

Evaluator Half Gate. For the evaluator half gate, $v_c = v_a \wedge v_b$, the evaluator knows the value of v_a when the gate is evaluated, and the generator knows neither input. Thus, the evaluator can behave differently depending on the known plaintext value of wire a. The generator provides the two ciphertexts:

$$H(a^0) \oplus c^0$$
$$H(a^1) \oplus c^0 \oplus b^0$$

The ciphertexts are not permuted here—since the evaluator already knows v_a, it is fine (and necessary) to arrange them deterministically in this order. When v_a is false, the evaluator knows it has a^0 and can compute $H(a^0)$ to obtain output wire c^0. When v_a is true, the evaluator knows it has a^1 so can compute $H(a^1)$ to obtain $c^0 \oplus b^0$. It can then xor this with the wire label it has for b, to obtain either c^0 (false, when $b = b^0$) or $c^1 = c^0 \oplus \Delta$ (true, when $b^1 = b^0 \oplus \Delta$),

without learning the semantic value of b or c. As with the generator half gate, using garbled row-reduction (Section 4.1.1) reduces the two ciphertexts to a single ciphertext. In this case, the generator simply sets $c^0 = H(a^0)$ (making the first ciphertext all zeroes) and sends the second ciphertext.

Combining Half Gates. It remains to show how the two half gates can be used to evaluate a gate $v_c = v_a \wedge v_b$, in a garbled circuit, where neither party can know the semantic value of either input. The trick is for the generator to generate a uniformly random bit r, and to transform the original AND gate into two half gates involving r:

$$v_c = (v_a \wedge r) \oplus (v_a \wedge (r \oplus v_b))$$

This has the same value as $v_a \wedge v_b$ since it distributes to $v_a \wedge (r \oplus r \oplus v_b)$. The first AND gate $(v_a \wedge r)$ can be garbled with a generator half-gate. The second AND gate $(v_a \wedge (r \oplus v_b))$ can be garbled with an evaluator half-gate, but only if $r \oplus v_b$ is leaked to the evaluator. Since r is uniformly random and not known to the evaluator, this leaks no sensitive information to the evaluator. The generator does not know v_b, but can convey $r \oplus v_b$ to the evaluator without any overhead, as follows. The generator will choose r to be the point-and-permute pointer bit of the false wire label on wire b, which is already chosen uniformly randomly. Thus, the evaluator learns $r \oplus v_b$ directly from the pointer bit on the wire it holds for b without learning anything about v_b.

Since the XOR gates do not require generating and sending garbled tables by using FreeXOR, we can compute an AND gate with only two ciphertexts, two invocations of H, and two "free" XOR operations. Zahur *et al.* (2015) proved the security of the half-gates scheme for any H that satisfies correlation robustness for naturally derived keys. In all settings, including low-latency local area networks, both the time and energy cost of bandwidth far exceed the cost of computing the encryptions (see the next section for how H is computed in this and other garbling schemes), and hence the half-gates method is preferred over any other known garbling scheme. Zahur *et al.* (2015) proved that no garbling scheme in a certain natural class of "linear" schemes[1] could

[1] See Zahur *et al.* (2015) for a precise formulation of this class. Roughly speaking, the half-gates scheme is optimal among schemes that are allowed to call a random oracle and perform fixed linear operations on wire labels / garbled gate information / oracle outputs, where the choice of these operations depends only on standard point-and-permute (Section 3.1.1) bits.

use fewer than two ciphertexts per gate. Hence, under these assumptions the half-gates scheme is bandwidth-optimal for circuits composed of two-input binary gates (see Section 4.5 for progress on alternatives).

4.1.4 Garbling Operation

Network bandwidth is the main cost for garbled circuits protocols in most practical scenarios. However, computation cost of GC is also substantial, and is dominated by calls to the encryption function implementing the random oracle H in garbling gates, introduced in Section 3.1.2. Several techniques have been developed to reduce that cost, in particular by taking advantage of built-in cryptographic operations in modern processors.

Since 2010, Intel cores have included special-purpose AES-NI instructions for implementing AES encryption, and most processors from other vendors include similar instructions. Further, once an AES key is set up (which involves AES round keys generation), the AES encryption is particularly fast. This combination of incentives motivated Bellare *et al.* (2013) to develop fixed-key AES garbling schemes, where H is implemented using fixed-key AES as a cryptographic permutation.

Their design is based on a *dual-key cipher* (Bellare *et al.*, 2012), where two keys are both needed to decrypt a ciphertext. Bellare *et al.* (2012) show how a secure dual-key cipher can be built using a single fixed-key AES operation under the assumption that fixed-key AES is effectively a random permutation. Since the permutation is invertible, it is necessary to combine the permutation with the key using the Davies-Meyer construction (Winternitz, 1984): $\rho(K) = \pi(K) \oplus K$. Bellare *et al.* (2013) explored the space of secure garbling functions constructed from a fixed-key permutation, and found the fastest garbling method using $\pi(K||T)[1:k] \oplus K \oplus X$ where $K \leftarrow 2A \oplus 4B$, A and B are the wire keys, T is a tweak, and X is the output wire.

Gueron *et al.* (2015) pointed out that the assumption that fixed-key AES behaves like a random permutation is non-standard and may be questionable in practice (Biryukov *et al.*, 2009; Knudsen and Rijmen, 2007). They developed a fast garbling scheme based only on the more standard assumption that AES is a pseudorandom function. In particular, they showed that most of the performance benefits of fixed-key AES can be obtained just by carefully pipelining the AES key schedule in the processor.

Note also that the FreeXOR optimization also requires stronger than standard assumptions (Choi *et al.*, 2012b), and the half-gates method depends on FreeXOR. Gueron *et al.* (2015) showed a garbling construction alternative to FreeXOR that requires only standard assumptions, but requires a single ciphertext for each XOR gate. Moreover, their construction is compatible with a scheme for reducing the number of ciphertexts needed for AND gates to two (without relying on FreeXOR, as is necessary for half gates). The resulting scheme has higher cost than the half-gates scheme because of the need to transmit one ciphertext for each XOR, but shows that it is possible to develop efficient (within about a factor of two of the cost of half gates) garbling schemes based only on standard assumptions.

4.2 Optimizing Circuits

Since the main cost of executing a circuit-based MPC protocol scales linearly with the size of the circuit, any reduction in circuit size will have a direct impact on the cost of the protocol. Many projects have sought ways to reduce the sizes of circuits for MPC. Here, we discuss a few examples.

4.2.1 Manual Design

Several projects have manually designed circuits to minimize the costs of secure computation (Kolesnikov and Schneider, 2008b; Kolesnikov *et al.*, 2009; Pinkas *et al.*, 2009; Sadeghi *et al.*, 2010; Huang *et al.*, 2011b; Huang *et al.*, 2012a), often focusing on reducing the number of non-free gates when FreeXOR is used. Manual circuit design can take advantage of opportunities that are not found by automated tools, but because of the effort required to manually design circuits, is only suitable for widely-used circuits. We discuss one illustrative example next; similar approaches have been used to design common building blocks optimized for secure computation such as multiplexers and adders, as well as more complex functions like AES (Pinkas *et al.*, 2009; Huang *et al.*, 2011b; Damgård *et al.*, 2012a).

Oblivious permutation. Shuffling an array of data in an oblivious permutation is an important building block for many privacy-preserving algorithms, including private set intersection (Huang *et al.*, 2012a) and square-root ORAM

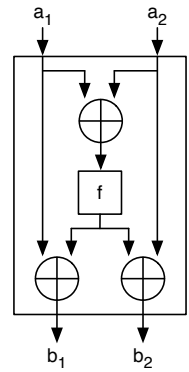

Figure 4.2: X switching block

(Section 5.4). A basic component of an oblivious permutation, as well as many other algorithms, is a *conditional swapper* (also called an *X switching block* by Kolesnikov and Schneider (2008b) and Kolesnikov and Schneider (2008a)), which takes in two inputs, a_1 and a_2, and produces two outputs, b_1 and b_2. Depending on the value of the swap bit p, either the outputs match the inputs ($b_1 = a_1$ and $b_2 = a_2$) or the outputs are the inputs in swapped order ($b_1 = a_2$ and $b_2 = b_1$). The swap bit p is known to the circuit generator, but must not be revealed to the evaluator. Kolesnikov and Schneider (2008b) provided a design for a swapper that takes advantage of FreeXOR and requires only a two-row garbled table (which can be reduced to a single ciphertext using garbled row reduction from Section 4.1.1). The swapper is implemented as:

$$\begin{aligned} b_1 &= a_1 \oplus (p \wedge (a_1 \oplus a_2)) \\ b_2 &= a_2 \oplus (p \wedge (a_1 \oplus a_2)) \end{aligned} \tag{4.1}$$

The swapper is illustrated in Figure 4.2. There the block f is set to 0 if no swapping is desired, and to 1 to implement swapping. The f block is implemented as a conjunction of the input with the programming bit p, so Figure 4.2 corresponds to Equation 4.1.

Since wire outputs can be reused, $p \wedge (a_1 \oplus a_2)$ only needs to be evaluated once. Referring back to the half-gates garbling and the notation of Section 4.1.3, when p is known to the generator, this conjunction is a generator half gate. As noted above, applying GRR3 allows this to be implemented with a single

ciphertext.[2] With the above conditional swapper, a random oblivious permutation can be produced by the circuit generator by selecting a random permutation and configuring the swappers in a Waksman network (Waksman, 1968) as necessary to produce it (Huang *et al.*, 2012a). Several permutation block designs, including truncated permutation blocks, are presented by Kolesnikov and Schneider (2008a).

Low-depth circuits: optimizing for GMW. For most of this chapter, we focus on Yao's GC, where a single round of communication is sufficient and the cost of execution scales with the size of the circuit. While most of the circuit-based optimizations apply to other protocols also, it is important to consider variations in cost factors when designing circuits for other protocols. In particular, in the GMW protocol (Section 3.2) each AND gate evaluation requires an OT, and hence a round of communication. AND gates on the same level can be batched and executed in the same round. Thus, unlike Yao's GC where the cost of execution is independent of its depth, for GMW protocol executions the cost depends heavily on the depth of the circuit.

Choi *et al.* (2012a) built an efficient implementation of GMW by using OT precomputation (Beaver, 1995) to reduce the on-line computation to a simple XOR, and OT extension protocols (Section 3.7.2) to reduce the overall cost of OT. A communication round (two messages) is still required during evaluation for each level of AND gates, however, so circuit execution time depends on the depth of the circuit. Schneider and Zohner (2013) provided further optimizations to the OT protocols for use in GMW, and designed low-depth circuits for several specific problems. Since GMW supports FreeXOR, the effective depth of a circuit is the maximum number of AND gates on any path. Schneider and Zohner (2013) were able to produce results on low-latency networks with GMW that were competitive with Yao's GC implementations by designing low-depth circuits for addition, squaring, comparison, and computing Hamming weight, and by using single-instruction multiple data (SIMD) instructions to pack operations on multiple bits, following on the approach used by Sharemind (Bogdanov *et al.*, 2008a).

[2] Indeed, the idea for half-gates garbling (Section 4.1.3) came from this X switching block design from Kolesnikov and Schneider (2008b).

4.2.2 Automated Tools

Boolean circuits go back to the earliest days of computing (Shannon, 1937). Because they have core applications in computing (e.g., in hardware components), there are a number of tools that have been developed to produce efficient Boolean circuits. The output of some of these tools can be adapted to circuit-based secure computation.

CBMC-GC. Holzer *et al.* (2012) used a model checking tool as the basis for a tool that compiles C programs into Boolean circuits for use in a garbled circuits protocol. CBMC (Clarke *et al.*, 2004) is a bounded model checker designed to verify properties of programs written in ANSI C. It works by first translating an input program (with assertions that define the properties to check) into a Boolean formula, and then using a SAT solver to test the satisfiability of that formula. CBMC operates at the level of bits in the machine, so the Boolean formula it generates is consistent with the program semantics at the bit level. When used as a model checker, CBMC attempts to find a satisfying assignment of the Boolean formula corresponding to the input program. If a satisfying assignment is found, it corresponds to a program trace that violates an assertion in the program. CBMC unrolls loops and inlines recursive function calls up to the given model-checking bound, removing cycles from the program. For many programs, CBMC can statically determine the maximum number of loop iterations; when it cannot, programmers can use annotations to state this explicitly. When used in bounded model checking, an assertion is inserted that will be violated if the unrolling was insufficient. Variables are replaced by bit vectors of the appropriate size, and the program is converted to single-static assignment form so that fresh variables are introduced instead of assigning to a given variable more than once.

Normally, CBMC would convert the program to a Boolean formula, but internally it is represented as a circuit. Hence, CBMC can be used as a component in a garbled circuits compiler that translates an input program in C into a Boolean circuit. To build CBMC-GC, Holzer *et al.* (2012) modified CBMC to output a Boolean circuit which can be then executed in circuit-based secure computation framework (such as the one from Huang *et al.* (2011b), which was used by CBMC-GC). Since CBMC was designed to optimize circuits for producing Boolean formulas for SAT solvers, modifications were

done to produce better circuits for garbled circuit execution. In particular, XOR gates are preferred in GC execution due to the FreeXOR technique (whereas the corresponding costs in model checking favor AND gates). To minimize the number of non-free gates, Holzer *et al.* (2012) replaced the built-in circuits CBMC would use for operations like addition and comparison, with designs that minimize costs with free XOR gates.

TinyGarble. Another approach to generating circuits for MPC is to leverage the decades of effort that have been invested in hardware circuit synthesis tools. Hardware description language (HDL) synthesis tools transform a high-level description of an algorithm, which could be written in a programming language or common HDL language such as Verilog, into a Boolean circuit. A synthesis tool optimizes a circuit to minimize its size and cost, and then outputs a *netlist*, which is a straightforward description of a circuit as a list of logic gates with connected inputs and outputs.

Conventional hardware synthesis tools, however, do not generate circuits suitable for MPC protocols because they generate circuits that may have cycles and they are designed to optimize for different costs that are encountered the MPC execution (in particular, they assume the cost of gates based on hardware implementations). With TinyGarble, Songhori *et al.* (2015) overcame these problems in adapting circuit synthesis tools for generating circuits for MPC, and Yao's GC in particular.

The approach of TinyGarble is to use sequential logic in a garbled circuit. In a sequential circuit, instead of just connecting gates in a combinational way where each gate's outputs depend only on its inputs and no cycles are permitted, a circuit also can maintain state. In hardware, state would be stored in a physical memory element (such as a flip-flop), and updated with each clock cycle. To execute (generate and send) a garbled circuit, TinyGarble instead unrolls a sequential circuit, so the stored state is an additional input to the circuit, and new garbled gates are generated for each iteration. This means the representation is compact, even for large circuit executions, which allows performance improvement due to the ability to store the circuit in processor cache and avoid expensive RAM accesses. This method trades off a slight increase in the number of garbled gates to execute for a reduction in the size of the circuit representation.

In addition, TinyGarble uses a custom circuit synthesis library to enable the circuit synthesis tool to produce cost-efficient circuits for MPC. This includes a library of custom-designed circuits for common operations like a multiplexer, based on previous designs (Section 4.2.1). Another input to a circuit synthesis tool is a *technology library*, that describes the logic units available on the target platform and their costs and constraints, and use this to map a structural circuit to a gate-level netlist. To generate circuits that take advantage of FreeXOR, the custom library developed for TinyGarble sets the area of an XOR gate to 0, and the area of other gates to a cost that reflects the number of ciphertexts required. When the circuit synthesis tool is configured to minimize circuit area, this produces circuits with an optimized number of non-XOR gates.

Songhori *et al.* (2015) report a 67% reduction in the number of non-XOR gates in 1024-bit multiplication compared to automatically-generated circuits for same function. For a more diverse function set (implementing a MIPS processor), the circuit generation has modest performance improvement as compared to prior work (the synthesized MIPS CPU circuit reduces the number of non-XOR gates by less than 15% compared to a straightforward assembly of MIPS CPU from constituent blocks).

4.3 Protocol Execution

The main limit on early garbled circuit execution frameworks, starting with Fairplay (Malkhi *et al.*, 2004), is that they needed to generate and store the entire garbled circuit. Early on, researchers focused on the performance for smaller circuits and developed tools that naïvely generate and store the entire garbled circuit. This requires a huge amount of memory for all but trivial circuits, and limited the size of inputs and complexity of functions that could be computed securely. In this section, we discuss various improvements to the way MPC protocols are executed that have overcome these scaling issues and eliminated much of the overhead of circuit execution.

Pipelined Execution. Huang *et al.* (2011b) introduced *garbled circuit pipelining*, which eliminated the need for either party to ever store the entire garbled circuit. Instead of generating the full garbled circuit and then sending it, the circuit generation and evaluation phases are interleaved. Before the circuit execution begins, both parties instantiate the circuit structure, which is small

relative to the size of the full garbled circuit since it can reuse components and is made of normal gate representations instead of non-reusable garbled gates.

To execute the protocol, the generator produces garbled gates in an order that is determined by the topology of the circuit, and transmits the garbled tables to the evaluator as they are produced. As the client receives them, it associates each received garbled table with the corresponding gate of the circuit. Since the order of generating and evaluating the circuit is fixed according to the circuit (and must not depend on the parties' private inputs), keeping the two parties synchronized requires essentially no overhead. As it evaluates the circuit, the evaluator maintains a set of live wire labels and evaluates the received gates as soon as all their inputs are ready. This approach allows the storage for each gate to be reused after it is evaluated, resulting in much smaller memory footprint and greatly increased performance.

Compressing Circuits. Pipelining eliminates the need to store the entire garbled circuit, but still requires the full structural circuit to be instantiated. Sequential circuits, mentioned earlier in Section 4.2.2, are one way to overcome this by enabling the same circuit structure to be reused but require a particular approach to circuit synthesis and some additional overhead to maintain the sequential circuit state. Another approach, initiated by Kreuter *et al.* (2013), uses lazy generation from a circuit representation that supports bounded loops. Structural circuits are compactly encoded using a *Portable Circuit Format* (PCF), which is generated by a circuit compiler and interpreted as the protocol executes. The input to the the circuit compiler is intermediate-level stack machine bytecode output by the LCC front-end compiler (Fraser and Hanson, 1995), enabling the system to generate MPC protocols from different high-level programs. The circuit compiler takes in an intermediate-level description of a program to execute as an MPC, and outputs a compressed circuit representation. This representation is then used as the input to interpreters that execute the generator and evaluator for a Yao's GC protocol, although the same representation could be used for other interpreters to executed different circuit-based protocols.

The key insight enabling PCF's scalability is to evaluate loops without unrolling them by reusing the same circuit structure while having each party locally maintain the loop index. Thus, new garbled tables can be computed

as necessary for each loop execution, but the size of the circuit, and local memory needed, does not grow with the number of iterations. PCF represents Boolean circuits in a bytecode language where each input is a single bit, and the operations are simple Boolean gates. Additional operations are provided for duplicating wire values, and for making function calls (with a return stack) and indirect (only forward) jumps. Instructions that do not involve executing Boolean operators do not require any protocol operations, so can be implemented locally by each party. To support secure computation, garbled wire values are represented by unknown values, which cannot be used as the conditions for conditional branches. The PCF compiler implemented several optimizations to reduce the cost of the circuits, and was able to scale to circuits with billions of gates (e.g., over 42 billion gates, of which 15 billion were non-free, to compute 1024-bit RSA).

Mixed Protocols. Although generic MPC protocols such as Yao's GC and GMW can execute any function, there are often much more efficient ways to implement specific functions. For example, additively homomorphic encryption schemes (including Paillier (1999) and Damgård and Jurik (2001)) can perform large additions much more efficiently than can be done with Boolean circuits.

With homomorphic encryption, instead of jointly computing a function using a general-purpose protocol, P_1 encrypts its input and sends it to P_2. P_2 then uses the encryption homomorphism to compute (under encryption) a function on the encrypted input, and sends the encrypted result back to P_1. Unless the output of the homomorphic computation is the final MPC result, its plaintext value cannot be revealed. Kolesnikov *et al.* (2010) and Kolesnikov *et al.* (2013) describe a general mechanism for converting between homomorphically-encrypted and garbled GC values. The party that evaluates the homomorphic encryption, P_2, generates a random mask r, which is added to the output of the homomorphic encryption, $\mathsf{Enc}(x)$, before being sent to P_1 for decryption. Thus the value received by P_1 is $\mathsf{Enc}(x + r)$, which P_1 can decrypt to obtain $x + r$. To enter x into the GC evaluation, P_1 provides $x + r$ and P_2 provides r as their inputs into the GC. The garbled circuit performs the subtraction to cancel out the mask, producing a garbled representation of x. Several works have developed customized protocols for particular tasks that combine homomorphic encryption with generic MPC (Brickell *et al.*, 2007;

Huang *et al.*, 2011c; Nikolaenko *et al.*, 2013a; Nikolaenko *et al.*, 2013b).

The TASTY compiler (Henecka *et al.*, 2010) provides a language for describing protocols involving both homomorphic encryption and garbled circuits. It compiles a high-level description into a protocol combining garbled circuit and homomorphic encryption evaluation. The ABY (Arithmetic, Boolean, Yao) framework of Demmler *et al.* (2015) support Yao's garbled circuits and two forms of secret sharing: arithmetic sharings based on Beaver multiplication triples (Section 3.4) and Boolean sharings based on GMW (Section 3.2). It provides efficient methods for converting between the three secure encodings, and for describing a function that can be executed using a combination of the three protocols. Kerschbaum *et al.* (2014) developed automated methods for selecting which protocol performs best for different operations in a secure computation.

Outsourcing MPC. Although it is possible to run MPC protocols directly on low-power devices, such as smartphones (Huang *et al.*, 2011a), the high cost of bandwidth and the limited energy available for mobile devices makes it desirable to outsource the execution of an MPC protocol in a way that minimizes the resource needed for the end user device without compromising security. Several schemes have been proposed for off-loading most of the work of GC execution to an untrusted server including Salus (Kamara *et al.*, 2012) and (Jakobsen *et al.*, 2016).

We focus here on the scheme from Carter *et al.* (2016) (originally published earlier (Carter *et al.*, 2013)). This scheme targets the scenario where a mobile phone user wants to outsource the execution of an MPC protocol to a cloud service. The other party in the MPC is a server that has high bandwidth and computing resources, so the primary goal of the design is to make the bulk of the MPC execution be between the server and cloud service, rather than between the server and mobile phone client. The cloud service may be malicious, but it is assumed not to collude with any other party. It is a requirement that no information about either the inputs or outputs of the secure function evaluation are leaked to the cloud service. This security notion of a non-colluding cloud is formalized by Kamara *et al.* (2012). The Carter *et al.* (2016) protocol supports malicious security, building on several techniques, some of which we discuss in Section 6.1. To obtain the inputs with lower

resources from the client, the protocol uses an outsourced oblivious transfer protocol. To provide privacy of the outputs, a blinding circuit is added to the original circuit that masks the output with a random pad known only to the client and server. By moving the bulk of the garbled circuit execution cost to the cloud service, the costs for the mobile device can be dramatically reduced.

4.4 Programming Tools

Many programming tools have been developed for building privacy-preserving applications using MPC. These tools vary by the input languages they support, how they combine the input program into a circuit and how the output is represented, as well as the protocols they support. Table 4.2 provides a high-level summary of selected tools for building MPC applications. We don't attempt to provide a full survey of MPC programming tools here, but describe one example of a secure computation programming framework next.

Obliv-C. The Obliv-C language is a strict extension of C that supports all C features, along with new data types and control structures to support data-oblivious programs that will be implemented using MPC protocols. Obliv-C is designed to provide high-level programming abstractions while exposing the essential data-oblivious nature of such computations. This allows programmers to implement libraries for data-oblivious computation that include low-level optimizations without needing to specify circuits.

In Obliv-C, a datatype declared with the **obliv** type modifier is oblivious to the program execution. It is represented in encrypted form during the protocol execution, so nothing in the program execution can depend on its semantic value. The only way any values derived from secret data can be converted back to a semantic value is by calling an explicit reveal function. When this function is invoked by both parties on the same variable, the value is decrypted by the executing protocol, and its actual value is now available to the program.

Control flow of a program cannot depend on oblivious data since its semantic value is not available to the execution. Instead, Obliv-C provides oblivious control structures. For example, consider the following statement where x and y are **obliv** variables:

```
obliv if (x > y) x = y;
```

Since the truth value of the x > y condition will not be known even at runtime, there is no way for the execution to know if the assignment occurs. Instead, every assignment statement inside an oblivious conditional context must use "multiplexer" circuits that select based on the semantic value of the comparison condition within the MPC whether to perform the update or have no effect. Within the encrypted protocol, the correct semantics are implemented to ensure semantic values are updated only on the branch that would actually be executed based on the oblivious condition. The program executing the protocol (or an analyst reviewing its execution) cannot determine which path was actually executed since all of the values are encrypted within the MPC.

Updating a cleartext value z within an oblivious conditional branch would not leak any information, but would provide unexpected results since the update would occur regardless of whether or not the oblivious conditional is true. Obliv-C's type system protects programmers from mistakes where non-**obliv** values are updated in conditional contexts. Note that the type checking is not necessary for security since the security of the **obliv** values is enforced at runtime by the MPC protocol. It only exists to help the programmers avoid mistakes by providing compile time errors for non-sensical code.

To implement low-level libraries and optimizations, however, it is useful for programmers to escape that type system. Obliv-C provides an unconditional block construct that can be used within an oblivious context but contains code that executes unconditionally. Figure 4.3 shows an example of how an unconditional block (denoted with ~**obliv**(var)) can be used to implement oblivious data structures in Obliv-C. This is an excerpt of an implementation of a simple resizable array implemented using a **struct** that contains oblivious variables representing the content and actual size of the array, and an opaque variable representing its maximum possible size. While the current length of the array is unknown (since we might append() while inside an **obliv if**), we can still use an unconditional block to track a conservative upper bound of the length. We use this variable to allocate memory space for an extra element when it might be needed.

This simple example illustrates how Obliv-C can be used to implement low-level optimizations for complex oblivious data structures, without needing to implement them at the level of circuits. Obliv-C has been used to implement libraries for data-oblivious data structures supporting random access memory

```
typedef struct {
    obliv int *arr;
    obliv int sz;
    int maxsz;
} Resizeable;

void writeArray(Resizeable *r, obliv int index, obliv int val) obliv;

// obliv function, may be called from inside oblivious conditional context
void append(Resizable *r, obliv int val) obliv {
    ~obliv(_c) {
        r→arr = reallocateMem(r→arr, r→maxsz + 1);
        r→maxsz++;
    }
    writeArray(r, r→sz, val);
    r→sz++;
}
```

Figure 4.3: Example use of an unconditional block (extracted from Zahur and Evans (2015)).

including Square-Root ORAM (Section 5.4) and Floram (Section 5.5), and to implement some of the largest generic MPC applications to date including stable matching at the scale needed for the national medical residency match (Doerner *et al.*, 2016), an encrypted email spam detector (Gupta *et al.*, 2017), and a commercial MPC spreadsheet (Calctopia, Inc., 2017).

4.5 Further Reading

Many methods for improving garbling have been proposed beyond the ones covered in Section 4.1. As mentioned in Section 4.1.3, the half-gates scheme is bandwidth optimal under certain assumptions. Researchers have explored several ways to reduce bandwidth by relaxing those assumptions including garbling schemes that are not strictly "linear" in the sense considered in the optimality proof (Kempka *et al.*, 2016), using high fan-in gates (Ball *et al.*, 2016) and larger lookup tables (Dessouky *et al.*, 2017; Kennedy *et al.*, 2017). MPC protocols are inherently parallelizable, but additional circuit design effort may be helpful for maximizing the benefits of parallel execution (Buescher and Katzenbeisser, 2015). GPUs provide further opportunities for speeding up

Tool / Input Language	Output/Execution	Protocols
ABY / Custom low-level Demmler *et al.*, 2015	Virtual machine executes protocol	Arithmetic, Boolean sharings; GC (§4.3)
EMP / C++ Library Wang *et al.*, 2017a	Compiled to executable	Authenticated Garbling (§6.7), others
Frigate / Custom (C-like) Mood *et al.*, 2016	Interprets compact Boolean circuit	Yao's GC, malicious-secure with DUPLO
Obliv-C / C + extensions Zahur and Evans, 2015	Source-to-source (C)	Yao's GC, Dual Execution (§7.6)
PICCO / C + extensions Zhang *et al.*, 2013	Source-to-source (C)	3+-party secret-sharing

Table 4.2: Selected MPC Programming tools. In this table, we focus on tools that are recently or actively developed, and that provide state-of-the-art performance. The DUPLO extension is from (Kolesnikov *et al.*, 2017b). All of the listed tools are available as open source code: ABY at https://github.com/encryptogroup/ABY; EMP at https://github.com/emp-toolkit; Frigate at https://bitbucket.org/bmood/frigaterelease; Obliv-C at https://oblivc.org; PICCO at https://github.com/PICCO-Team/picco.

MPC execution (Husted *et al.*, 2013).

Many other MPC programming tools have been developed. Wysteria (Rastogi *et al.*, 2014) provides a type system that supports programs that combine local and secure computation. SCAPI (Bar-Ilan Center for Research in Applied Cryptography and Cyber Security, 2014; Ejgenberg *et al.*, 2012) provides Java implementations of many secure computation protocols. We focused on tools mostly building on garbled circuit protocols, but many tools implement other protocols. For example, the SCALE-MAMBA system (Aly *et al.*, 2018) compiles programs written in a custom Python-like language (MAMBA) to execute both the offline and online phases of secure computation protocols built on BDOZ and SPDZ (Section 6.6.2). We focused on programming tools in the dishonest majority setting, but numerous tools have been built supporting other threat models. In particular, very efficient implementations are possible with assuming three-party, honest-majority model, most notably Sharemind (Bogdanov *et al.*, 2008b) and Araki *et al.* (2017) (Section 7.1.2).