Optimizing Semi-Honest Secure Multiparty Computation for the Internet

Aner Ben-Efraim[*]
Dept. of Computer Science
Ben-Gurion University, Israel
anermosh@post.bgu.ac.il

Yehuda Lindell[†]
Dept. of Computer Science
Bar-Ilan University, Israel
lindell@biu.ac.il

Eran Omri[‡]
Dept. of Computer Science
Ariel University
omrier@gmail.com

Abstract

In the setting of secure multiparty computation, a set of parties with private inputs wish to compute some function of their inputs without revealing anything but their output. Over the last decade, the efficiency of secure *two-party* computation has advanced in leaps and bounds, with speedups of some orders of magnitude, making it fast enough to be of use in practice. In contrast, progress on the case of multiparty computation (with more than two parties) has been much slower, with very little work being done. Currently, the only implemented efficient multiparty protocol has many rounds of communication (linear in the depth of the circuit being computed) and thus is not suited for Internet-like settings where latency is not very low.

In this paper, we construct highly efficient *constant-round* protocols for the setting of multiparty computation for semi-honest adversaries. Our protocols work by constructing a multiparty garbled circuit, as proposed in BMR (Beaver et al., STOC 1990). Our first protocol uses oblivious transfer and constitutes the *first* concretely-efficient constant-round multiparty protocol for the case of no honest majority. Our second protocol uses BGW, and is significantly more efficient than the FairplayMP protocol (Ben-David et al., CCS 2008) that also uses BGW.

We ran extensive experimentation comparing our different protocols with each other and with a highly-optimized implementation of semi-honest GMW. Due to our protocol being constant round, it significantly outperforms GMW in Internet-like settings. For example, with 13 parties situated in the Virginia and Ireland Amazon regions and the SHA256 circuit with 90,000 gates and of depth 4000, the overall running time of our protocol is 25 seconds compared to 335 seconds for GMW. Furthermore, our *online time* is under half a second compared to 330 seconds for GMW.

[*]Supported by Israel Science Foundation grant 544/13, the cyber security research center at Ben Gurion University, and by the Frankel center for computer science.

[†]Supported by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS) and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

[‡]Supported by a grant from the Israeli Science and Technology ministry and by Israel Science Foundation grant 544/13.

CCS'16, October 24-28, 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4139-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978347

1. INTRODUCTION

1.1 Background

In the setting of secure multiparty computation (MPC), a set of parties wish to compute a joint function of their private inputs without revealing anything but the output. Protocols for secure computation guarantee *privacy* (meaning that the protocol reveals nothing but the output) and *correctness* (meaning that the correct function is computed), in the presence of adversarial behavior. There are two classic adversary models that are typically considered: *semi-honest* (where the adversary follows the protocol specification but may try to learn more than is allowed from the protocol transcript) and *malicious* (where the adversary can run any arbitrary polynomial-time strategy in its attempt to breach security). In this paper, we focus on the multiparty setting and semi-honest adversaries. Our focus is on the construction of *concretely efficient* protocols for this case (where by "concrete efficiency" we mean faster run time in practice).

Two paradigms for secure computation. There are two main paradigms for constructing general secure two-party and multiparty computation protocols that can be used to securely compute any function. The GMW (Goldreich-Micali-Wigderson) paradigm [15] works by having the parties interact to compute every (AND) gate in the circuit (this approach is also followed for information-theoretic protocols and arithmetic circuits). Such protocols typically send very little bandwidth per gate, but suffer from multiple rounds of communication, being linear in the depth of the circuit. They are therefore very fast in LAN settings, but *very slow when communicating over the Internet.* In contrast, in the garbled-circuit paradigm [32] an entire circuit is constructed in a way that enables it to be evaluated without revealing anything but the output. The original construction by Yao [32] worked only for two parties, and the notion of a multiparty garbled circuit was later shown by BMR (Beaver-Micali-Rogaway) in [5]. Such protocols are typically slower in LAN settings since they require much more bandwidth, but have the potential to be *much faster on slower networks* like the Internet since they have a constant number of rounds.

Concrete efficiency and the last decade. Secure two-party computation has progressed in the past decade in leaps and bounds. It has progressed from a notion of theoretical interest only, into a technology that is even being commercialized by multiple companies. In order to demonstrate this progress, we go back to the first implementation of Yao's garbled circuits in Fairplay [29] in 2004. In this paper, they ran a secure computation on a "large circuit" with 4383 gates overall, with security in the presence of semi-honest adversaries. On a LAN, their protocol ran in 7.09 seconds. In 2014, it takes approximately 16 milliseconds to run an analogous execution on a circuit that is about 5 times the size with over 22,000 gates, of which 6800 are AND gates [17].[1] This therefore constitutes a multiplicative improvement of approximately 2000 (or over 3 orders of magnitude). Observe that Moore's law (the version saying that computing power doubles every 2 years) can only account for a speedup of a factor of 32. This amazing progress was due to a long series of works that focused on all cryptographic and algorithmic aspects of the problem, as well as the advent of ubiquitous crypto hardware acceleration in the form of AES-NI and more; see [25, 20, 18, 3, 6, 31, 33] for just some examples. The current situation is that we now have a very good understanding of the cost of secure two-party computation in practice.

We stress that not only did these works yield impressive progress for protocols that are secure for semi-honest adversaries, they also fueled progress for protocols that are secure for malicious adversaries. In particular, the techniques and methods used to speed up garbled circuit generation and evaluation were directly incorporated in protocols for malicious adversaries, and fast OT extensions for the semi-honest case served as a basis for fast OT extensions for the malicious case.

Analogously, the first implementation of a multiparty protocol (with more than two parties) was FairplayMP [7] in 2008, which followed the multiparty garbled circuit approach of [5]. The largest circuit they computed had 1024 gates, and the time it took to run the protocol ranged from approximately 10 seconds for 5 parties to 55 seconds for 10 parties. In 2012, [10] implemented the GMW protocol for the multiparty setting. For a circuit with 5,500 AND gates (about 5 times that of FairplayMP), they reported times ranging from approximately 7 seconds for 5 parties to about 10 seconds for 10 parties.[2] Surprisingly, to the best of our knowledge, there has been *no additional work on general (circuit-based) multiparty secure computation* for the case of semi-honest adversaries and no honest majority.[3] In particular, there have been no improvements to the multiparty garbled circuit approach since FairplayMP, and the time that it takes to run multiparty computations is orders of magnitude more than for two-party computations. Furthermore, the FairplayMP protocol is only secure for an honest majority. Thus, no concretely-efficient constant-round multiparty protocol for the case of no honest majority semi-honest adversaries has *ever* been described or implemented.

We remark that the *multiparty* setting is needed for many applications like auctions, trading, elections, privacy-preserving surveys and more. Thus, we believe that it is of great importance to remedy the current situation where efficient multiparty computation lags so far behind efficient two-party computation.

1.2 Our contributions

In this paper, we study the problem of multiparty computation with semi-honest adversaries. We have three main contributions:

1. We present the first *concretely-efficient* constant-round multiparty protocol that is secure for *any* number of corrupted parties. Our protocol is based on the multiparty garbled-circuit approach of [5], and requires each party to run an oblivious transfer with every other party per gate in order to construct the garbled circuit.

2. Following FairplayMP [7], we also present protocols for constructing a multiparty garbled circuit based on BGW [8] (BenOr-Goldwasser-Wigderson), that are secure for an honest majority. Our protocols are far more efficient than those presented in [7].

3. We ran extensive experiments comparing the performance of our different protocols to each other and to multiparty GMW. We ran experiments over 3 different networks, with different numbers of parties, and with different circuits. Our results deepen our understanding of the impact of round complexity on efficiency, and which protocols are suited to low and high latency networks.

Our protocols are all secure for semi-honest adversaries. Semi-honest security is sufficient when parties somewhat trust each other, but are concerned with inadvertent leakage or cannot share their raw information due to privacy regulations. It is also sufficient in cases where it is reasonable to assume that the parties running the protocol are unable to replace the installed code. Nevertheless, security against malicious adversaries is preferable, providing far higher guarantees. However, such protocols will always be far less efficient. In addition, we argue that the first step towards obtaining highly efficient protocols for malicious adversaries is to understand the semi-honest case and how it can be made very efficient. This was indeed the case for two-party computation, where optimizations and work carried out for the semi-honest case was an important factor towards obtaining efficient protocols for the case of malicious adversaries.

Before proceeding, we remark that the garbled-circuit approach in the multiparty case is fundamentally different to the two-party case and introduces many difficulties. In the two-party case, one party constructs the garbled circuit and the other evaluates it. Thus, the computation involved in constructing and evaluating the garbled circuit is *local*. In contrast, in the multiparty case, it is not possible for one party to construct the garbled circuit on its own. This is due to the fact that if that party colludes with one of the parties evaluating the circuit, then the parties' inputs will all be revealed (since the party who constructed the circuit knows all of the keys). Rather, it is necessary for all of the

[1]Indeed, we are comparing the total number of gates, since the invention of free-XOR is one of the factors involved in the progress that took place over this decade.

[2]The implementation of [10] is actually highly optimized and performs far better than reported. This is because they included the initial synchronization of communication in their times, which can include large delays unless done carefully. We use their code for our comparisons, but run the timer only once the actual MPC protocol begins.

[3]In contrast, work has been done for *malicious* adversaries; e.g. [12, 27, 28]. However, the basic semi-honest case has been skipped over, and along with it techniques and understanding that are extremely valuable for the malicious case as well.

parties to *collaboratively construct the circuit* via a secure protocol. Thus, an efficient instantiation of this approach requires **(a)** an efficient garbled-circuit approach, and **(b)** an efficient way of collaboratively generating it. Note that the construction of the garbled circuit (which is the more expensive part of the protocol) can be carried out in an offline phase. Then, once inputs are received, an online phase can be run with very minimal communication and just local evaluation of the multiparty garbled circuit.

We will now elaborate on each of our major contributions.

Dishonest majority. As we have mentioned, [7] designed a protocol for securely computing a multiparty garbled circuit using the BGW protocol. Since the construction of a garbled gate requires multiplication between a large value and a bit, this lends itself naturally to an arithmetic circuit approach. However, this introduces many complications in [7], especially since they work over a prime-order finite field. In contrast, we observe that multiplication of a string by a bit can actually be carried out with a single string oblivious transfer (OT). Furthermore, bit and string OT is very cheap today, due to the extremely fast OT extension protocols that exist that reach rates of approximately only one microsecond per oblivious transfer on a LAN and whatever the latency is on slower networks [20, 3]. In addition to our basic protocol, we show how approximately 1/4 of the OT cost can be removed. This uses a novel approach that enables us to compute one of the entries in each garbled gate by sending a single message and combining it with the results of the OTs used to compute the other entries in the garbled gate. We then further reduce the cost by utilizing the fact that we actually only need a variant of OT, called correlated OT, which is even more efficient [3]. Finally, we are the first to incorporate free-XOR [25] into a multiparty garbled circuit. Previous works did not utilize free-XOR [7, 28] since they work over a prime-order finite field. Needless to say, as in the two-party case, this optimization is crucial for obtaining high performance.

We remark that the incorporation of garbled-circuit optimizations that exist for the two-party case is not necessarily straightforward. This is because the circuit must itself be computed via a secure protocol. For example, we do not know how to efficiently perform garbled row-reduction when building a multiparty garbled circuit. This makes the half-gates optimization for Yao's garbled circuits [33] unsuitable since it is less efficient when there is no row reduction. We conjecture that row reduction (and thus half-gates) is not suitable for the multiparty setting.

Honest majority. The FairplayMP multiparty garbled-circuit construction used BGW and worked over a prime-order finite field. As we have mentioned, this has some disadvantages: first, it is not compatible with the free-XOR optimization; second it requires an additional multiplication to convert shares of values which may be $\{-1,+1\}$ into shares of bits in $\{0,1\}$. In contrast, we work over $GF(2^{128})$ which is of characteristic 2. This enables us to incorporate free-XOR and to save the multiplication to convert $\{-1,+1\}$ to $\{0,1\}$. As another benefit of using free-XOR, we were able to define a formula for computing each value in each garbled gate that saves an additional multiplication. Thus, [7] require 4 secure multiplications per element per gate inside BGW, whereas we only require 2 secure multiplications per element per gate inside BGW. Beyond computational costs, this reduces the round complexity from 6 to 4.

Experimental evaluations. We ran multiple experiments to compare our different constant-round protocols to each other as well as to a highly optimized implementation of GMW [10]. These are described in depth in Section 4. Our code is open-source and available in the SCAPI library [2] to enable others to reproduce our results and compare to future work.

Our results show that for computing deep circuits on Internet-like networks, our protocols way outperform GMW. For example, in executions ran on Amazon with machines communicating between Virginia and Ireland (with a 75ms round-trip) on the SHA256 circuit with 90,000 AND gates and depth 4,000, the GMW protocol took over 5 minutes (for 3 to 13 parties) whereas our protocol ranged from about 6 seconds for 3 parties to 25 seconds for 13 parties. Furthermore, the vast majority of the time in GMW is the online time whereas the vast majority of the time in our protocol is the offline time. Thus, we obtain online times ranging from 170ms to 455ms for 3 and 13 parties, respectively, while the online time for GMW is approximately 5 minutes.

To our surprise, our protocol that is secure without an honest majority and uses oblivious transfer is almost always more efficient than our protocols based on BGW. This was unexpected since oblivious transfer uses cryptographic operations, whereas BGW uses only simpler information-theoretic operations. However, a closer look shows that the number of field multiplications in BGW is actually cubic in the number of parties, whereas the cost in the oblivious-transfer protocol is quadratic. Combining this with the fact that the best oblivious transfer extension protocols today [3, 24, 22] are so fast, the BGW protocol only outperforms the oblivious transfer a small number of parties.

Another interesting outcome from our experiments is that the GMW protocol actually performs very well on low-depth circuits or in very fast networks. This is due to the fact that when the circuit is shallow then the number of rounds in GMW is also small. In this case, the small bandwidth is a big advantage over the garbled circuit approach which requires large message transmission. In addition, in fast networks with very low latency, GMW's many rounds does not significantly affect the running time.

Conclusions. In the setting of secure multiparty computation over the Internet (which is the natural setting for the aforementioned applications of auctions, elections, privacy-preserving surveys and so on), our protocol following the garbled circuit approach is much faster than all previous protocols. Furthermore, our protocol has an extremely fast online time, making it suitable for scenarios where preprocessing is possible and fast response time is needed.

It is important to also note that the complexity of GMW grows *linearly* in the number of parties, in contrast to the multiparty garbled-circuit approach which is *quadratic* in the number of parties (we count the complexity per party, and thus linear complexity for GMW means that *each* party's work is linear in the number of parties per gate). Thus, when the number of parties is very large, this cost can be significant. A very interesting open question coming out of our work is whether or not it is possible to construct and evaluate a multiparty garbled circuit in time that is linear in the number of parties, with concrete efficiency.

1.3 Related Work

As we have mentioned, there has been a long line of work optimizing secure two-party computation, both in the semi-honest and malicious settings; cf. [25, 20, 18, 3, 6, 31, 33]. The first protocol implemented for the multiparty setting with semi-honest adversaries was that of FairplayMP [7] in 2008. Later, in 2012, a highly optimized version of the multiparty GMW protocol for semi-honest adversaries was presented by [10]. This implementation uses oblivious transfer extensions [20], making the cost of the oblivious transfers insignificant. In addition, [10] run all of the oblivious transfers on random input in an offline phase, and then only need to send a single bit in each direction per oblivious transfer in the online phase. This makes the online phase very fast with very little bandwidth. For the specific case of three parties, the Sharemind protocol achieves fast computation [9], but only for 3 parties and only for an honest majority. In addition, their protocol also has many rounds, like GMW, and so is not suitable for Internet-like settings. The recent work of [30] achieves malicious security for 3 parties with an honest majority using Yao garbled-circuits, but does not extend to beyond 3 parties.

In addition to the work on semi-honest multiparty computation, there has been work on multiparty computation that is secure in the presence of *malicious* adversaries The SPDZ protocol [13, 12], with improvements in [23], is the state-of-the-art in this area. The offline phase of these works is over an order of magnitude slower than ours, which is fully justified by the fact that they achieve security for malicious adversaries which is much more difficult. More significantly, the SPDZ protocol's online time follows the GMW paradigm (but is somewhat more expensive due to the necessity to enforce correctness). Thus, in a slow network, it suffers from the same problems as the GMW protocol that we compare with here. Other protocols for the multiparty setting that focus on efficiency include [21] and [27]; however, these protocols have not been implemented and seem to be considerably slower in practice. The closest comparison to our protocol in the malicious setting is that of [28] which uses SPDZ in order to build a multiparty garbled circuit. Thus, [28] is expected to have an online time similar to that of our protocol. However, its offline time is estimated at over 400 seconds for 3 parties computing the AES circuit (over two orders of magnitude slower than ours). We stress that this discussion regarding protocols secure for malicious adversaries is not for the purpose of comparing our protocol (since we only achieve semi-honest security) but to stress that the semi-honest model yields far more efficient protocols and is thus important to study for applications where semi-honest security suffices. The comparison also highlights that much progress is needed in this area; our belief is that improvements for the semi-honest case will significantly help in the malicious case as well.

2. THE BMR PROTOCOL

2.1 Background

As we have mentioned, in the BMR protocol the parties construct a multiparty analog of Yao's garbled circuit. In Yao's garbled circuit construction, random labels (which are just keys) are assigned to each wire; one label represents the 0-value and the other label represents the 1-value. Then, each gate is constructed by encrypting the appropriate output-wire label with the appropriate input-wire label. For example, let g be an AND gate with input wires u, v and output wire w, and let κ denote the security parameter. Furthermore, let $k_{u,0}, k_{u,1} \in \{0,1\}^\kappa$ denote the labels on wire u, and likewise for wires v and w. Then, $k_{w,0}$ is double encrypted under $k_{u,0}, k_{v,0}$, under $k_{u,0}, k_{v,1}$, and under $k_{u,1}, k_{v,0}$, while $k_{w,1}$ is double encrypted under $k_{u,1}, k_{v,1}$. Observe that given a single label on each input wire, it is possible to compute the *correct* output-wire label by decrypting the single ciphertext that can be decrypted with one label on each input wire. Furthermore, by providing these ciphertexts in random order, the party computing the gate has no idea whether it obtained $k_{w,0}$ or $k_{w,1}$ (since they are both just random values). Thus, the party evaluating the circuit learns nothing at all from the computation. In the setting of two parties, one party prepares the garbled circuit and the other evaluates it.

In the multiparty setting, all the parties must prepare the garbled circuit together, and it must have the property that no subset of colluding parties can learn anything about what value is being computed. The multiparty garbled circuit is therefore constructed by having all parties contribute to the garbling of every gate. In particular, all n parties choose their own random 0-labels and random 1-labels on every wire, and the output wire labels are encrypted separately under every single party's input wire labels. Thus, a single honest party's input-wire labels hide the output-wire labels from all other parties. Let $k^i_{u,b}$ be the input-label that party P_i holds for value $b \in \{0,1\}$ on wire u; likewise for wires v and w. Since there are n parties and n output-wire labels for each value $b \in \{0,1\}$, the encryption works by masking all of $k^1_{w,0}, \ldots, k^n_{w,0}$ with each pair $(k^i_{u,0}, k^i_{v,0})$, $(k^i_{u,0}, k^i_{v,1})$ and $(k^i_{u,1}, k^i_{v,0})$. To be more exact, let F^2 denote a *double-key pseudorandom function* that takes two keys k, k' and maintains security as long as at least one key is secret. Abusing notation and denoting the gate function by $g(\cdot,\cdot)$ (as well as the gate index by g), we have that for every $a, b \in \{0,1\}$ the output labels of all parties $P_1, \ldots, P_n$ associated with the bit $g(a,b)$ are encrypted by

$$\left\{ \left(\bigoplus_{i=1}^{n} F^2_{k^i_{u,a}, k^i_{v,b}}(g \circ j) \right) \oplus k^j_{w,g(a,b)} \right\}_{j=1}^{n} \qquad (1)$$

where $\circ$ denotes string concatenation. Observe that the keys on the input wires of *all* of the parties are needed in order to learn $k^1_{w,g(a,b)}, \ldots, k^n_{w,g(a,b)}$ (the concatenation of all keys/labels is called the "superseed" in [5, 7]). We remark that the value input to the PRF is the gate number followed by the index of the label being masked. This ensures that all output-wire labels are masked with independent values.

In order to hide which values of a and b are being dealt with, the "point-and-permute technique" (that was originally invented by BMR) is used. According to this technique, a random secret "permutation bit" λ_u is associated with every wire u (this is achieved by each party P_i choosing a random λ^i_u and setting $\lambda_u = \oplus^n_{i=1} \lambda^i_u$). Then, if the actual bits on inputs wire u and v to gate g are a and b, then the parties see $\lambda_u \oplus a$ and $\lambda_v \oplus b$, and this guides them as to which ciphertext to decrypt. In addition, they obtain the output $\lambda_w \oplus g(a,b)$ that enables them to proceed to the next gate.

Constructing the multiparty garbled circuit. In the first part of the protocol, the parties run a secure protocol to compute the garbled circuit. This phase is independent of the parties' actual inputs and so can therefore be run in a separate *offline phase*. The important property that is the "BMR claim-to-fame" is the fact that the entire circuit can be constructed in a constant number of rounds, independently of the depth of the circuit. This is due to the fact that the circuit computing a multiparty garbled gate is very shallow, and all garbled gates can be computed in parallel (since given the garbled labels on all wires, each gate can be independently computed). Thus, it is possible to use *any* protocol for general secure computation with round complexity linear in the depth of the circuit in order to securely compute a multiparty garbled circuit in a constant number of rounds (irrespective of its depth). Intuitively, each party P_i locally computes $F^2_{k^i_{u,a},k^i_{v,b}}(g \circ j)$ for every $a, b \in \{0,1\}$ and $j \in [n]$, and these are input to a circuit that computes the gate. Observe that this means that each party must carry out $4n$ double-key PRF computations per gate.

Evaluating the BMR circuit. As with Yao's garbled circuits, given the sets of n keys $k^1_{u,a}, \ldots, k^n_{u,a}$ and $k^1_{v,b}, \ldots, k^n_{v,b}$, it is possible to decrypt a single row in the garbled gate and obtain $k^1_{w,g(a,b)}, \ldots, k^n_{w,g(a,b)}$. This method can therefore be used to evaluate the entire circuit. Furthermore, once the parties receive the sets of keys on every circuit-input wire, they can each locally compute the entire circuit and obtain the output. Thus, the online running-time of the protocol is very small: parties send a very small amount of information, and then the entire circuit is just locally computed. Observe that in order to carry out the gate computation, each party needs to compute $F^2_{k^i_{u,a},k^i_{v,b}}(g \circ 1), \ldots, F^2_{k^i_{u,a},k^i_{v,b}}(g \circ n)$, for $i = 1, \ldots, n$. Thus, evaluation of the multiparty garbled circuit requires n^2 PRF computations per gate.

2.2 Free XOR

In this section, we describe how we incorporated the free-XOR optimization into the multiparty garbled circuit. The original work by BMR [5], and the FairplayMP [7] system were both published before the free-XOR technique [25]. This technique enables XOR gates to be computed for free in Yao's garbled circuits, and has become standard for use in that setting. In the recent BMR-SPDZ protocol of [28] that uses BMR to achieve security in the presence of *malicious adversaries*, free XOR was also not incorporated. This seems to be due to the fact that they work in a finite field of prime order (in order to optimize the SPDZ portion of the protocol) and this does not lend itself to free XOR. Our first optimization in this work is to incorporate free-XOR into BMR. This is important for two reasons. First, and most obviously, free-XOR reduces the number of AES operations in garbling and evaluating the circuit. Since evaluation requires n^2 AES operations per gate (for n parties), this is significant. Second, free-XOR significantly *reduces the bandwidth* of the garbled circuit since only AND gates require encryptions. The large amount of communication in BMR is problematic and so this is important.

Intuitively, the free XOR in BMR works in the same way as in Yao. For every $i = 1, \ldots, n$, a fixed (random) difference parameter $R^i \in \{0,1\}^\kappa$ is chosen (known only to P_i). Then, for every wire w, the label $k^i_{w,0}$ is random whereas the label $k^1_{w,1}$ is set to equal $k^i_{w,0} \oplus R^i$. Due to this relation, XOR gates can be computed by simply XORing the garbled labels on the input wires, and taking the garbled label on the output wire to be the result. We formally describe the functionality that outputs a BMR circuit in Figure 1; this functionality needs to be securely computed in the offline phase. We design the functionality in a way that minimizes the computation necessary within the secure computation protocol (thus, all shares of all values are prepared by parties before running $\mathcal{F}_{GC}$).

Functionality $\mathcal{F}_{GC}$ for Constructing a Multiparty Garbled Circuit

Inputs: All parties hold the circuit C, the number of parties n, and the security parameter κ. In addition, each party P_i has the following private inputs:

1. A global difference string $R^i \in \{0,1\}^\kappa$.
2. A share $\lambda^i_w \in \{0,1\}$ of the permutation bit for every wire w.
3. Its parts $k^i_{w,0}, k^i_{w,1} \in \{0,1\}^\kappa$ of the garbled labels for every wire $w \in W$.

   (As described in Section 2.3 (Figure 2), the $\lambda^i_w \in \{0,1\}$ and $k^i_{w,0}, k^i_{w,1} \in \{0,1\}^\kappa$ values are chosen in a special way to enable free XOR.)

Computation: The functionality computes the garbled circuit GC. For every AND gate $g \in C$ with input wires u, v and output wire w, every $\alpha, \beta \in \{0,1\}$, and every $j \in [n]$, compute:

$$\tilde{g}^j_{\alpha,\beta} = \left(\bigoplus_{i=1}^{n} F^2_{k^i_{u,\alpha},k^i_{v,\beta}}(g \circ j) \right) \oplus k^j_{w,0} \oplus \Big(R^j \cdot ((\lambda_u \oplus \alpha) \cdot (\lambda_v \oplus \beta) \oplus \lambda_w) \Big) \qquad (2)$$

Outputs: The functionality outputs $\left(\tilde{g}^1_{\alpha,\beta} \circ \cdots \circ \tilde{g}^n_{\alpha,\beta} \right)$ for every g and every $\alpha, \beta \in \{0,1\}$ to all parties.

Figure 1: Functionality $\mathcal{F}_{GC}$

We explain the computation of $\tilde{g}^j_{\alpha,\beta}$ in Figure 1. The permutation bits $\lambda_u, \lambda_v, \lambda_w$ randomly permute the values on each wire. In particular, the parties see α, β on the input wires, but the *actual value* on wires u and v are $\alpha \oplus \lambda_u$ and $\beta \oplus \lambda_v$, respectively. (Since λ_u, λ_v are random, it follows that α, β reveal *nothing* about the actual values, as required.) Now, it follows that the output of the AND gate is exactly $(\lambda_u \oplus \alpha) \cdot (\lambda_v \oplus \beta)$ since this is the product of the actual values. Assume for a moment that $\lambda_w = 0$. Then, $\tilde{g}^j_{\alpha,\beta}$ will be an encryption of $k^j_{w,0}$ if $(\lambda_u \oplus \alpha) \cdot (\lambda_v \oplus \beta) = 0$, and otherwise it will be an encryption of $k^j_{w,0} \oplus R^j = k^j_{w,1}$, as required. In contrast, if $\lambda_w = 1$, then the result will be reversed. Thus, this preserves the invariant that the parties hold the key $k^j_{w,b}$ where b equals the XOR of the actual value on the wire with λ_w.

In the full version of this paper, we formally prove security of the free-XOR construction in the context of BMR, under a correlation-robust circularity assumption as in [11] (which holds for a random oracle).

2.3 MPC Using Multiparty Garbled Circuits

Given the functionality $\mathcal{F}_{GC}$, we now show how any multiparty functionality can be securely computed in a constant number of rounds. We have already explained informally how the protocol works in Section 2.1, and we now present the formal specification of the offline protocol and of the online phases. These specifications assume an implementation of $\mathcal{F}_{GC}$ which is the main protocol challenge, as we will see below in Section 3.

The MPC protocol that we describe is a variant of the BMR protocol and is designed in the $\mathcal{F}_{GC}$-hybrid model. The main difference between this variant and the original BMR protocol is the fact that our BMR circuit incorporates free-XOR. (Of course, our full protocol has additional differences in the efficient computation of $\mathcal{F}_{GC}$.) The protocol consists of two phases:

Part 1 - the Offline Phase: In this phase, the parties first run a local computation to prepare their inputs to $\mathcal{F}_{GC}$. A formal specification of this local computation is given in Figure 2. Observe that the parties choose the wire labels in a special way in order to ensure that the free-XOR property works. Next, the parties securely compute Functionality $\mathcal{F}_{GC}$ in order to obtain the garbled circuit. Finally, each party P_i sends its shares of the permutation bits λ_w on every circuit-output wire w. In addition, all parties send P_i their shares of the permutation bits on the circuit-input wires associated with P_i's private input. (Recall that a formal specification of Functionality $\mathcal{F}_{GC}$ appears above in Figure 1.)

The BMR Offline Phase

1. Each P_i chooses a random key offset $R^i \in_R \{0,1\}^\kappa$.
2. For every wire w that is *not* the output of a XOR gate, each party P_i works as follows:
   (a) P_i chooses a random bit λ_w^i as a share of λ_w.
   (b) P_i chooses its part of the 0-garbled label $k_{w,0}^i \in \{0,1\}^\kappa$ at random.
   (c) P_i sets its part of the 1-garbled label to be $k_{w,1}^i = k_{w,0}^i \oplus R^i$.
3. For every wire w that *is* the output of a XOR gate (in topological order over the circut) with input wires u, v and output wire w, each party P_i works as follows:
   (a) P_i sets its share of the permutation bit $\lambda_w^i = \lambda_u^i \oplus \lambda_v^i$.
   (b) P_i sets its part of the garbled labels on wire w as $k_{w,0}^i = k_{u,0}^i \oplus k_{v,0}^i$, and $k_{w,1}^i = k_{w,0}^i \oplus R^i$.
4. Each party P_i calls $\mathcal{F}_{GC}$ with the inputs R^i, and $\lambda_w^i, k_{w,0}^i, k_{w,1}^i$ for every wire w. The parties receive back all garbled gates as output. (This step involves running one of the protocols described in Section 3 for securely computing $\mathcal{F}_{GC}$.)
5. For every *circuit-output* wire w, each P_i sends λ_w^i to all other parties, who all compute $\lambda_w = \oplus_{i=1}^n \lambda_w$.
6. For every *circuit-input* wire w, if P_i's is the party whose input is associated with wire w, then each P_j sends λ_w^j to P_i. Party P_i computes $\lambda_w = \oplus_{i=1}^n \lambda_w$. As in [7], this step can be skipped by having all parties P_j with $j \neq i$ set $\lambda_w^j = 0$ for all *circuit-input wires* w.

Figure 2: Offline phase – BMR circuit construction

Part 2 - the Online Phase: Given their private inputs, this phase begins with the parties broadcasting the XOR of their private input with the permutation bit on the associated wire. Next, the other parties send the appropriate garbled labels. Finally, each party locally computes the garbled circuit and obtains the output (using knowledge of λ_w on the output wire to convert the output garbled value into an actual output bit). Thus, the online phase consists of very little communication and primarily local computation. A formal specification of this phase is given in Figure 3.

The BMR Online Phase

1. **Send garbled labels associated with inputs:** For every circuit-input wire w:
   (a) Let P_i be the party whose input is associated with wire w and let x_{i_w} be P_i's input bit associated with the wire. Then, P_i sends $\alpha_w = x_{i_w} \oplus \lambda_w$ to all parties, where $\lambda_w = \oplus_{i=1}^n \lambda_w^i$. For every wire w, we denote by α_w the XOR of the actual value on the wire (based on the input) and λ_w; we call this the public value.
   (b) Each party P_j sends its part k_{w,α_w}^j of the garbled label on w to all other parties.
   (c) At this point, each party holds $k_{w,\alpha_w}^1, \ldots, k_{w,\alpha_w}^n$ for every circuit-input wire.
2. **Local circuit computation:** Each party locally evaluates the garbled circuit. The circuit is evaluated by traversing the circuit in topological order, computing the garbled labels on the output wire of each gate one at a time.
   Each gate is computed as follows. Let g be the current gate with input wires u, v and output wire w. Let α and β be the *public values* on wires u and v, respectively.
   (a) If g is a XOR gate, then each party computes the public value on the output wire w to be $\gamma = \alpha \oplus \beta$. In addition, for every $j = 1, \ldots, n$ it computes $k_{w,\gamma}^j = k_{u,\alpha}^j \oplus k_{v,\beta}^j$.
   (b) If g is an AND gate, then each party computes
   $$k_{w,\gamma}^j = \tilde{g}_{\alpha,\beta}^j \oplus \left(\bigoplus_{i=1}^n F^2_{k_{u,\alpha}^i, k_{v,\beta}^i}(g \circ j) \right)$$
   for every $j \in [n]$, and for α, β which are the public values on wires u, v as above. Given $k_{w,\gamma}^1, \ldots, k_{w,\gamma}^n$, each party P_i compares the ith value to the garbled labels $k_{w,0}^i, k_{w,1}^i$ that it input in the offline phase on this wire. If it equals $k_{w,0}^i$ then it sets the public value on wire w to be $\gamma = 0$; otherwise it sets $\gamma = 1$.
3. **Output determination:** For every output wire w, each party computes the *actual* output bit of wire w to be $\alpha_w \oplus \lambda_w$, where α_w is the *public value* on wire w and λ_w is as received in the offline phase.

Figure 3: Online phase – BMR circuit evaluation

The following theorem follows directly from the security of the multiparty garbled circuit and how to use it, as proven in [5] (with our addition of free-XOR), and from the fact that $\mathcal{F}_{GC}$ securely computes the circuit:

THEOREM 2.1. *Let f be an n-party functionality. Then, the Protocol in Figures 2 and 3 securely computes f in the presence of a semi-honest adversary corrupting any number of parties.*

2.4 The Double Key PRF

Both the offline and the online phases use the double-key PRF F^2. We implemented F^2 in two ways: **(1)** by concatenating the two 128-bit keys and using it as a a single AES-256 key, as proposed by [26], **(2)** using fixed-key AES, as proposed by [6]. Since the online time is dominated by the AES computations (with very little communication), fixed-key AES is considerably faster than the first method, even when AES-NI is used in both. For the offline times, the difference between these 2 methods was minor.

3. SECURELY COMPUTING $\mathcal{F}_{GC}$

It is clear that the online phase of the protocol is fast, as it is essentially n^2 times the cost of locally computing a standard Yao garbled circuit, where the latter is known to be extremely fast [6]. Thus, the main challenge in making the protocol concretely efficient is in constructing an efficient protocol that securely computes the multiparty garbled circuit. Stated differently, the goal is to construct an efficient protocol that securely computes the $\mathcal{F}_{GC}$ functionality.

The original BMR protocol [5] considered an honest majority of parties, although this is only needed for achieving security in the presence of malicious adversaries. In any case, [5] did not specify a concrete protocol for securely computing $\mathcal{F}_{GC}$ (or, more exactly their version of it), but rather stated that any secure protocol can be used. In [7], they use the BGW protocol [8] for an honest majority in order to securely compute $\mathcal{F}_{GC}$. Surprisingly, no *concretely efficient* protocol has been suggested to securely compute $\mathcal{F}_{GC}$ without an honest majority. We will present two main protocols here: **(1)** a protocol for a dishonest majority using oblivious transfer (OT) as a black box, **(2)** several variants of the protocol based on BGW (with significant protocol improvements over [7]).

3.1 Protocol for No Honest Majority

In this section, we present a protocol that securely computes $\mathcal{F}_{GC}$ in the OT-hybrid model, in the presence of semi-honest adversaries. As we have mentioned, this is the *first* concretely-efficient constant-round protocol for this setting. Our protocol securely computes Eq. (2) in Figure 1 for every $j = 1, \ldots, n$ and all four choices of $\alpha, \beta \in \{0, 1\}$. Intuitively, in order to compute Eq. (2), the parties will need to generate shares of $R^j \cdot ((\lambda_u \oplus \alpha) \cdot (\lambda_v \oplus \beta) \oplus \lambda_w)$ for all j. Observe that P_j holds R_j, and the other values are all shared (i.e., each party holds λ_u^i and $\lambda_u = \oplus_{i=1}^n \lambda_u^i$, and so on). As we will see, this computation can be reduced to a computation whereby parties securely compute XOR shares of the product of two parties' input bits.

3.1.1 *Secure Multiplication*

Secure bit-bit multiplication. The main operation that we use is the secure computation of XOR shares of the product of input bits. That is, we define the functionality $f_\times(a, b) = (c, d)$ where $c \oplus d = a \cdot b$. The folklore protocol for computing $f_\times$ works as follows:

1. Parties P_1 and P_2 hold input bits $a \in \{0, 1\}$ and $b \in \{0, 1\}$, respectively.

2. P_1 chooses a random $r \in \{0, 1\}$. P_1 sets $x_0 = r$ and $x_1 = r \oplus a$.

3. P_1 and P_2 run a *bit* oblivious transfer where P_1 plays the sender with inputs (x_0, x_1) and P_2 plays the receiver with input bit b. Party P_2 receives x_b.

4. P_1 outputs r and P_2 outputs x_b.

Observe that $x_b = r \oplus a \cdot b$ (since $b = 0$ implies that $x_b = x_0 = r$ and $b = 1$ implies that $x_b = x_1 = r \oplus a$). Thus, $r \oplus x_b = a \cdot b$, as required. The cost of securely computing $f_\times$ is therefore a *single* OT. The protocol is trivially secure since the only information received is via the OT; thus P_1 learns nothing and P_2 learns only its share of the output.

Secure string-bit multiplication. Observe that within Eq. (2), we need to also multiply R^j by a bit. It is possible to use $f_\times$ for this. However, since R^j is of length κ (in practice, say 128), this increases the number of oblivious transfers by κ, which is very significant. We therefore would like to securely compute the functionality $f_\times^\kappa(s, b) = (c, d)$ where $c \oplus d = s \cdot b$ and where $s \in \{0, 1\}^\kappa$ and $b \in \{0, 1\}$. Fortunately, $f_\times^\kappa$ can be securely computed using a *single string OT* in the exact same way as $f_\times$ with P_1 inputting $(r, r \oplus s)$ in the oblivious transfer. Thus, this also costs a single oblivious transfer.

3.1.2 *The Protocol for $\mathcal{F}_{GC}$*

We describe our protocol in the $(f_\times, f_\times^\kappa)$-hybrid model where the parties have access to a trusted party computing $f_\times$ and $f_\times^\kappa$ for them. As we have seen, these functionalities can be securely computed efficiently using oblivious transfer.

PROTOCOL 3.1 (SECURELY COMPUTING *Eq.* (2)).

1. **Step 1:** In this step, the parties securely compute XOR shares of $\lambda_{uv} \stackrel{\text{def}}{=} \lambda_u \cdot \lambda_v$. Recalling that each party P_i holds λ_u^i, λ_v^i, we have that
$$\begin{aligned}\lambda_{uv} &= \lambda_u \cdot \lambda_v = \left(\oplus_{i=1}^n \lambda_u^i\right) \cdot \left(\oplus_{i=1}^n \lambda_u^i\right) \\ &= \left(\oplus_{i=1}^n \lambda_u^i \cdot \lambda_v^i\right) \oplus \left(\oplus_{i \neq j} \lambda_u^i \cdot \lambda_v^j\right). \qquad (3)\end{aligned}$$
Each party P_i can locally compute $\lambda_u^i \cdot \lambda_v^i$. In addition, each pair of parties P_i, P_j (with $i \neq j$) runs $f_\times(\lambda_u^i, \lambda_v^j)$ and $f_\times(\lambda_u^j, \lambda_v^i)$ in order to obtain XOR shares of the products. Finally, each P_i XORs $\lambda_u^i \cdot \lambda_v^i$ together with all of the output shares received from $f_\times$; denote the result by λ_{uv}^i. By Eq. (3), it follows that $\oplus_{i=1}^n \lambda_{uv}^i = \lambda_{uv}$, as required.

2. **Step 2:** Given shares of $\lambda_{uv} = \lambda_u \cdot \lambda_v$, the parties generate XOR shares of $\lambda_{uvw} \stackrel{\text{def}}{=} \lambda_{uv} \oplus \lambda_w$, of $\lambda_{u\bar{v}w} \stackrel{\text{def}}{=} \lambda_u \cdot \bar{\lambda}_v \oplus \lambda_w$, of $\lambda_{\bar{u}vw} \stackrel{\text{def}}{=} \bar{\lambda}_u \cdot \lambda_v \oplus \lambda_w$, and of $\lambda_{\bar{u}\bar{v}w} \stackrel{\text{def}}{=} \bar{\lambda}_u \cdot \bar{\lambda}_v \oplus \lambda_w$. This can be carried out using local computation only.

   Trivially, in order to obtain shares of λ_{uvw} it suffices for each P_i to compute $\lambda_{uvw}^i = \lambda_{uv}^i \oplus \lambda_w^i$ (where λ_w^i is its input share on the output wire).

   Furthermore, observe that $\lambda_u \cdot \bar{\lambda}_v = \lambda_u \cdot (\lambda_v \oplus 1) = \lambda_u \cdot \lambda_v \oplus \lambda_u$. Thus, each P_i can compute its share $\lambda_{u\bar{v}w}^i = \lambda_{uv}^i \oplus \lambda_u^i \oplus \lambda_w^i$. In a similar way, each P_i computes its shares $\lambda_{\bar{u}vw}^i = \lambda_{uv}^i \oplus \lambda_v^i \oplus \lambda_w^i$ and $\lambda_{\bar{u}\bar{v}w}^i = \lambda_{uv}^i \oplus \lambda_u^i \oplus \lambda_v^i \oplus \lambda_w^i$ (where P_1 XORs its result with 1 in this last case).

3. **Step 3:** In this step, the parties securely compute XOR shares of $R^j \cdot ((\lambda_u \oplus \alpha) \cdot (\lambda_v \oplus \beta) \oplus \lambda_w)$, for every $j = 1, \ldots, n$ and every $\alpha, \beta \in \{0, 1\}$. Observe that they already have XOR shares of $\lambda_{uvw}, \lambda_{u\bar{v}w}, \lambda_{\bar{u}vw}, \lambda_{\bar{u}\bar{v}w}$, and so need only to run $f_\times^\kappa$. That is, for every $j = 1, \ldots, n$, party P_j runs four invocations of $f_\times^\kappa$ with every P_i. In every invocation P_j inputs R^j, whereas P_i inputs its share λ_{uvw}^i in the first invocation, $\lambda_{u\bar{v}w}^i$ in the second invocation, and so on. Denote by $\rho_{j,\alpha,\beta}^i$ party P_i's share of $R^j \cdot ((\lambda_u \oplus \alpha) \cdot (\lambda_v \oplus \beta) \oplus \lambda_w)$.

4. **Step 4:** In this final step, the parties conclude the secure computation of Eq. (2); recall that the aim is for *each* party to hold the $4n$ values $\{\tilde{g}_{0,0}^j, \tilde{g}_{0,1}^j, \tilde{g}_{1,0}^j, \tilde{g}_{1,1}^j\}_{j=1}^n$. We describe this for general α, β and use the shares of $\rho_{j,\alpha,\beta}^i$ that were generated above.

For every $j = 1, \ldots, n$ and every $\alpha, \beta \in \{0,1\}$, party P_j sends $F^2_{k^j_{u,\alpha}, k^j_{v,\beta}}(g \circ j) \oplus k^j_{w,0} \oplus \rho^j_{j,\alpha,\beta}$ to all other parties, and each party P_i sends the string $F^2_{k^i_{u,\alpha}, k^i_{v,\beta}}(g \circ j) \oplus \rho^i_{j,\alpha,\beta}$ to all other parties. The parties then XOR all these values together, and the result is $\tilde{g}^j_{\alpha,\beta}$, as required.

Security. Intuitively, the protocol is secure since in Steps 1–3 the only values that the parties see are random shares output from $f_\times$ and $f^\kappa_\times$. Then, in Step 4, the parties all send their shares to each other. However, these shares are all random under the constraint that their XOR is $\tilde{g}^j_{\alpha,\beta}$, and so reveal nothing but the result. The full proof of the following will appear in the full version of this paper.

Theorem 3.2. *Protocol 3.1 securely computes the n-party $\mathcal{F}_{GC}$ functionality with n parties in the $(f_\times, f^\kappa_\times)$-hybrid model, in the presence of a semi-honest adversary corrupting any number of parties.*

Cost. Using the OT extension protocol of [20, 3], we have that each (bit or string) oblivious transfer costs two hash function invocations and two rounds of communication (a single message from the receiver to the sender and a single message back). However, bit-OT is significantly cheaper than string-OT since it requires much less communication, which is the bottleneck. Note that bit-OT is much faster than string-OT, but a single string-OT for strings of length κ is much faster than running κ executions of bit-OT.

We count the number of OT calls run by each party (as sender and as receiver); recall that $f_\times$ is just a single bit-OT and $f^\kappa_\times$ is just a single string-OT. The number of OT invocations equals $2(n-1)$ bit-OT plus $8(n-1)$ string-OT: in Step 1 each party computes $2(n-1)$ bit-OTs, and in Step 3 each party computes $8(n-1)$ string-OTs ($4(n-1)$ as sender and $4(n-1)$ as receiver). Observe that the use of $f^\kappa_\times$ instead of κ calls to $f_\times$ is very significant and reduces the cost in Step 3 from $8(n-1)\kappa$ bit-OTs to $8(n-1)$ string-OTs. The number of rounds in the entire protocol is five; two rounds in each of Step 1 and 3, and a single message in Step 4.

3.1.3 Reducing the Number of String-OTs

We now show that it is possible to reduce the number of string-OTs from $8(n-1)$ to $6(n-1)$, saving 25%. Specifically, we show how it is possible to securely compute $\tilde{g}^j_{1,1}$ from the shares $\rho^i_{j,0,0}, \rho^i_{j,0,1}, \rho^i_{j,1,0}$ (see Step 3 of the protocol) without running any OTs. As a result, the parties do *not* need to run the $f^\kappa_\times$ invocations where the receiver inputs its share of $\lambda^i_{\bar{u}\bar{v}w}$. This means that each P_j only needs to run three invocations of $f^\kappa_\times$ as sender, instead of four.

In order to compute $\tilde{g}^j_{1,1}$, party P_j sends the message $F^2_{k^j_{u,1}, k^j_{v,1}}(g \circ j) \oplus \left(k^j_{w,0} \oplus R^j\right) \oplus \rho^j_{j,0,0} \oplus \rho^j_{j,0,1} \oplus \rho^j_{j,1,0}$ to all other parties, and each party P_i sends the string $F^2_{k^i_{u,1}, k^i_{v,1}}(g \circ j) \oplus \rho^i_{j,0,0} \oplus \rho^i_{j,0,1} \oplus \rho^i_{j,1,0}$ to all other parties. We argue that the XOR of all these values is $\tilde{g}^j_{1,1}$. In order to see this, first denote $\oplus^n_{i=1} F^2_{k^i_{u,1}, k^i_{v,1}}(g \circ j)$ by $K_{g,j}$. Then, we have that the XOR of all these values equals

$$K_{g,j} \oplus k^j_{w,0} \oplus R^j \oplus R^j \cdot (\lambda_u \cdot \lambda_v \oplus \lambda_w)$$
$$\oplus R^j \cdot \left(\bar{\lambda}_u \cdot \lambda_v \oplus \lambda_w\right) \oplus R^j \cdot \left(\lambda_u \cdot \bar{\lambda}_v \oplus \lambda_w\right).$$

Rewriting the above, we have that it equals $K_{g,j} \oplus k^j_{w,0}$ $\oplus R^j \cdot (1 \oplus \lambda_u \cdot \lambda_v \oplus \bar{\lambda}_u \cdot \lambda_v \oplus \lambda_u \cdot \bar{\lambda}_v \oplus \lambda_w)$. (Note that λ_w appears 3 times and so 2 cancel out.) Now, for any bits $x, y \in \{0,1\}$, it holds that $1 \oplus x \cdot y \oplus \bar{x} \cdot y \oplus x \cdot \bar{y} = \bar{x} \cdot \bar{y}$. Thus, we conclude that the XOR of all the values equals $K_{g,j} \oplus k^j_{w,0} \oplus R^j \cdot \left(\bar{\lambda}_u \cdot \bar{\lambda}_v \oplus \lambda_w\right)$ which is exactly $\tilde{g}^j_{1,1}$.

The security of this optimization holds since all messages sent are masked by $F^2_{k^i_{u,1}, k^i_{v,1}}(g \circ j)$ and therefore reveal nothing. (They look random and so can be easily simulated.)

3.1.4 Using More Efficient OT

It was shown in [3] that in the Yao protocol, it is possible to use a variant of OT – called *correlated OT* – that is significantly more efficient. This version of OT can be used when there is a difference Δ and the sender's input is a random pair (x_0, x_1) with $x_0 \oplus x_1 = \Delta$. When using the free-XOR method within Yao's protocol, it follows that the garbled labels on the input wires can be random under the constraint that they have a fixed difference. Thus, correlated-OT suffices. Observe now that all the calls to OT are for computing $f_\times$ and $f^\kappa_\times$. In these protocols, party P_1 with input bit a (resp., string s) runs the OT-sender with *random inputs* (x_0, x_1) under the constraint that $x_0 \oplus x_1 = a$ (resp., $x_0 \oplus x_1 = s$). The protocols are not described in that way, but this is the result of choosing r and setting $x_0 = r$ and $x_1 = r \oplus a$. Thus, it is possible to use the correlated-OT protocol here and save additional cost.

3.2 A New BGW-Based Protocol

The FairplayMP [7] system used the information-theoretic BGW protocol [8] in order to securely compute the $\mathcal{F}_{GC}$ functionality. In this section, we present a protocol that also uses the BGW protocol in order to securely compute $\mathcal{F}_{GC}$. However, there are a number of important differences:

1. We point out and fix a small bug in the FairplayMP protocol: In FairplayMP, each party singlehandedly chooses all the keys for its input wires. This renders the protocol insecure: every party is able to compute the circuit multiple times, with different inputs of its own (with a fixed setting of the other parties' inputs).

2. FairplayMP works over a prime-order field. In contrast, we work in $GF(2^k)$ which is a field of characteristic 2. First and foremost, this makes it possible to incorporate the free-XOR technique into BMR. Second, by using a field of characteristic 2, we are able to reduce the round complexity of the protocol. This is due to the fact that in such a field the XOR of λ_{uv} with λ_w (as required when computing Eq. (2)) can be carried out with local computation only. In contrast, FairplayMP requires interaction to compute this (XOR is not addition in their finite field, and so they need a protocol to compute this).

3. We use a finite field that is large enough to embed each individual key $k^j_{u,\alpha}$, whereas FairplayMP embed the entire "superseed" $k^1_{u,\alpha}, \ldots, k^n_{u,\alpha}$ of length $n \cdot \kappa$ in the finite field. In the case of 33 parties and keys of length 128 bits, FairplayMP would have to work in a finite field with elements of size at least 4,224 bits. This means that FairplayMP scales poorly for large numbers of parties (since each operation becomes more expensive). In contrast, we work with each key separately and so the finite field is always of length κ, resulting in linear growth in the cost per party (at least with respect to the basic operations over $k^1_{u,\alpha}, \ldots, k^n_{u,\alpha}$).

4. We compute multiplications of the λ combinations only with $R^1, \ldots, R^n$. In contrast, FairplayMP compute their analog of Eq. (2) by multiplying both $k^1_{u,0} \circ \cdots \circ k^n_{u,0}$ and $k^1_{u,1} \circ \cdots \circ k^n_{u,1}$ with the λ combinations.

As we will see below, the focus of our protocol design here was to *reduce the number of rounds to an absolute minimum.*

Our basic BGW-based protocol. In general, BGW can be used to compute *any* arithmetic circuit. Furthermore, addition is free in the circuit, and multiplication requires interaction (and computation). When working in a finite field of characteristic 2, it follows that *XOR equals addition*, making the computation of Eq. (2) trivial. Specifically, Eq. (2) simply defines an arithmetic circuit over the field with two multiplications and $4n$ additions. To be exact, in order to compute $\tilde{g}^j_{\alpha,\beta}$:

1. Party P_j inputs values K_j, R^j, U_j, V_j, W_j where $K_j = F^2_{k^j_{u,\alpha}, k^j_{v,\beta}} \oplus k^j_{w,0}$, $U_j = \lambda^j_u \oplus \alpha$, $V_j = \lambda^j_v \oplus \beta$ and $W_j = \lambda^j_w$.
2. Each party P_i $(i \neq j)$ inputs K_i, U_i, V_i, W_i where $K_i = F^2_{k^i_{u,\alpha}, k^i_{v,\beta}}$, $U_i = \lambda^i_u$, $V_i = \lambda^i_v$ and $W_i = \lambda^i_w$.
3. Using BGW, the parties evaluate the circuit for: $(\oplus^n_{i=1} K_i) \oplus R^j \cdot ((\oplus^n_{i=1} U_i) \cdot (\oplus^n_{i=1} V_i) \oplus (\oplus^n_{i=1} W_i))$.

Beyond the ability to use free-XOR, we therefore see that the choice of the field to be of characteristic 2 yields a much simpler protocol as well. Since BGW is secure, we have that this protocol for computing $\mathcal{F}_{GC}$ is automatically secure.

Protocol details and round complexity – BGW4. The above BGW protocol can be carried out in just 4 rounds of communication; for this reason we call it BGW4 (to distinguish it from other variants that we will describe below). In order to see how 4 rounds suffice, we describe its steps in detail. All $\tilde{g}^j_{\alpha,\beta}$ values (for all $j = 1, \ldots, n$, all $\alpha, \beta \in \{0,1\}$ and all gates) are computed in parallel. Each value is computed as follows:

Communication round 1: All inputs are shared using t-out-of-n Shamir secret sharing, with $t = \lceil n/2 \rceil - 1$.

Local computation: The parties carry out local additions, as required by the circuit, in order to obtain shares of $\lambda_u \oplus \alpha = \oplus^n_{i=1} U_i$ and $\lambda_v \oplus \beta = \oplus^n_{i=1} V_i$. In addition, the parties locally multiply their shares of $\lambda_u \oplus \alpha$ and $\lambda_v \oplus \beta$. Denote this product by $\lambda^{\alpha,\beta}_{u,v}$.

Communication round 2: The parties run the GRR [14] degree reduction step on the product $\lambda^{\alpha,\beta}_{u,v}$ to recover legitimate shares of $\lambda^{\alpha,\beta}_{u,v}$ of degree t. This is best achieved by each party i resharing $c_i \cdot \lambda^{\alpha,\beta}_{u,v}$ in a t-out-of-n sharing, where c_i is a fixed (degree-reduction) constant of that party (which is precomputed once), and then summing the received shares.

Local computation: The parties locally add their shares of λ_w to $\lambda^{\alpha,\beta}_{u,v}$. Next, the parties once again locally multiply their shares of $\lambda^{\alpha,\beta}_{u,v} \oplus \lambda_w$ with R^j.

Communication round 3: The parties run the GRR degree reduction step on $R^j \cdot (\lambda^{\alpha,\beta}_{u,v} \oplus \lambda_w)$ as above.

Local computation: The parties locally add the shares of the K_i values to the result of the previous step.

Communication round 4: The parties send their shares to each others and reconstruct to obtain $\tilde{g}^j_{\alpha,\beta}$.

Computational complexity. The above is repeated $4n$ times for each gate with each computation requiring 3 Shamir secret sharings. Each sharing requires evaluating a degree-t polynomial on n points, which takes $t \cdot n \approx n^2/2$ field multiplications. The number of multiplications is therefore dominated by the first and third communication rounds, where the number of shares is $\approx 4n$ per gate, thus requiring $\approx 2n^3$ multiplications per gate. Therefore, asymptotically, the parties compute $4n^3$ field multiplications per gate of the circuit (concretely, there are slightly more), which is *cubic* in the number of parties. As we will see in Section 4, this significantly affect performance when the number of parties is not small. (We stress that the number of multiplications would be only quadratic had we worked in a field of size $n \cdot \kappa$. However, since multiplication is quadratic, the cost of computing one multiplication in a field of size $n \cdot \kappa$ is $n^2 \cdot \kappa^2$, versus a cost of $n \cdot \kappa^2$ for n multiplications in a field of size κ.)

3.3 Reducing the Round Complexity

In this section, we briefly describe how the number of rounds can be further reduced. The idea of reducing rounds by omitting reduction steps in BGW already appeared in [19], but the following is a more concretely efficient way of achieving this.

BGW3 – three rounds with an honest majority. The protocol described above that is secure with an honest majority has 4 rounds of communication. However, it is possible to save one round of communication by simply not running the degree reduction step on the $R^j \cdot (\lambda^{\alpha,\beta}_{u,v} \oplus \lambda_w)$ values. This works since the parties only add their shares of the K_i values to $R^j \cdot (\lambda^{\alpha,\beta}_{u,v} \oplus \lambda_w)$. Thus, if the K_i values are shared using a $2t$-degree polynomial instead (at the onset), then the degrees of the shares of K_i and $R^j \cdot (\lambda^{\alpha,\beta}_{u,v} \oplus \lambda_w)$ will be equal, and so they can be locally added. The reconstruction works in the same way, and thus this yields a 3-round protocol. We remark that beyond the fact that this has one round less, it also has *far less communication.* This is because the second degree-reduction is carried out on the shares of $R^j \cdot (\lambda^{\alpha,\beta}_{u,v})$ and so there are $4n$ degree-reductions per gate, resulting in a large amount of bandwidth. Thus, as we will see in Section 4, this protocol is *always better*, and often significantly better, than the 4-round protocol BGW4.

BGW2 – two rounds with more than 2/3 honest. It is possible to further reduce the round complexity of the protocol to just *two rounds* in the case that more than 2/3 of the parties are guaranteed to be honest, since no degree reduction is needed at all. However, this has less effect on the amount of communication (since there are only a small number of degree reductions remaining).

3.4 Optimizing Field Multiplications

We use $\kappa = 128$ and therefore work in $GF(2^{128})$. We use the Intel PCLMULQDQ carry-less multiplication instruction to obtain extremely fast pipelined multiplication operations in $GF(2^{128})$ [16]. This works by combining 4 carry-less multiplication operations over 64-bit operands. In addition, observe that the majority of the multiplications are for generating Shamir secret sharing. This works by taking a distinct field element ω_i for every party, and evaluating random polynomials on these values (which means multiplying by $\omega_i, \omega_i^2, \ldots, \omega_i^d$). By choosing ω_i to be of small degree, we have that for a small enough d, all of $\omega_i, \omega_i^2, \ldots, \omega_i^d$ are less than 64 bits long, and so a 128-bit element is multiplied with a 64-bit element. This can be achieved in just 2 carry-less multiplications instead of 4. When the number of parties exceeds 15, at least 5 bits are needed for each ω_i value and so the length of ω_i^d exceeds 64 bits, making the optimization no longer applicable.

4. EXPERIMENTAL EVALUATION

We implemented our protocols and ran multiple experiments for different circuit sizes and depths, different number of parties, and different latencies. In order to obtain a deeper understanding of the strengths and (possible) weaknesses of the BMR protocol, we included the GMW protocol in our experiments. The GMW protocol has multiple rounds of communication, but far less bandwidth than the BMR protocol. In addition, it is currently the only other general protocol for multiparty computation for the case of dishonest majority known. We used the code of [10] for these experiments. As described in Section 1.3, this code is highly optimized and uses OT extension in an offline phase for precomputing the oblivious transfers. We ran their code on the same circuits and platforms as our BMR code. (For the sake of brevity, in this section we refer to "BMR" as our different protocols; it is not to be misinterpreted as the original protocol of [5] which was purely theoretical.)

We measured times separately for the offline and online portion of the executions. In BMR, the entire construction of the circuit can be run offline, before the inputs are given. Thus, the online phase has very little communication and is dominated by local computation of the circuit. In contrast, in the GMW protocol, the offline phase consists only of preprocessing the oblivious transfers. We stress that the code of [10] does not merely run the base OTs as part of the OT extensions in the offline phase. Rather, they run *random OTs* and use the OT preprocessing method of [4]. Thus, in the online phase, the receiver sends a single bit to the sender and the sender replies with two bits. In addition, the parties entire computation requires merely XORing the preprocessed bits with the actual inputs. As a result, the GMW online phase has very little computation and communication, but many rounds of communication. We stress that in applications where low latency is required, the online time of a protocol is the main factor to be considered (assuming reasonable offline time). In contrast, in cases where the overall throughput is what is important, the total time is the main factor to be considered.

Platforms and parameters. We ran our experiments on three different platforms. First, we ran them on a cluster with a very low latency network called CREATE [1] (which is part of the DETER project). The cluster is comprised of Intel XEON 2.20 GHz machines (E5-2420) with 6 cores running Linux (Ubuntu1404-64-STD), and with a 1Gb connection and ping time between computers of approximately 0.1ms. In addition, we ran our experiments on Amazon c4.8xlarge instances (running Windows Server 2012 R2 Base) in two configurations: in the first, all machines are located in a single zone (Virginia); in the second, the machines were split between Ireland and Virginia (in preliminary experiments, this gave similar performance to where the latency was the same between all parties). The ping time within a single region is approximately 1ms, and the ping time between Virginia and Ireland is approximately 75ms. The bandwidth of all connections (according to Amazon's specifications) is 10Gb. The experiments in CREATE provide a very clean (low-latency) environment without external network effects, whereas the experiments in Amazon reflect performance in more "real-world" settings with networking fluctuation.

We ran the protocol for 3, 5, 7, 9, 11 and 13 parties, and in some cases 33 parties. We used 2 different circuits: a circuit computing the AES function with 6,800 AND gates and 256 input wires per party, and a circuit computing SHA256 with 90,825 AND gates and 512 input wires per party. Since the depth of the circuit is very significant for GMW, we also generated 4 synthetic circuits of the same size as the SHA256 circuit, but with respective depths of $d = 10$, $d = 100$, $d = 1000$ and $d = 4000$ (the real SHA256 circuit has depth 4000).

Raw results. Each experiment was run 50 times, and we computed the mean and standard deviation of all runs. The numbers in the tables here are measured in milliseconds, and present a 95% confidence interval. Observe that the interval is sometimes very large. This is especially true in Amazon in Virginia-Ireland. This expresses the reality that when sending a large amount of data over the Internet, the fluctuations are significant. The full tables containing all results will appear in the full version of this paper.

4.1 Protocol Improvements Measurements

We measured the effect of the different protocol optimizations on the running time. We ran the basic OT-based protocol without additional optimizations (Section 3.1.2), with the reduced number of OTs (Section 3.1.3), using correlated-OT (Section 3.1.4), with the free-XOR optimization, and with two threads between every pair of parties (increasing to more than two threads between every pair of parties reduced performance since the number of threads becomes very high with many parties). Our measurements were made on the CREATE platform, with 13 parties and the SHA-256 circuit with 90,825 AND gates and 42,029 XOR gates. The results are as follows:

	Basic	Red. OT	C-OT	Free-XOR	2 OT-Threads
OT-Off	6434	5549 ± 207	4011 ± 156	2851 ± 101	2633 ± 117
OT-On		399 ± 46	434 ± 36	313 ± 18	311 ± 20

The gain by using a reduced number of OTs and by using the correlated-OT protocol is very significant. In all cases, free-XOR provides exactly the expected speedup by dropping the cost by the ratio of XOR gates to overall gates.

Since all the improvements are also related to communication, we also tested them in a network with network latency of 76 ms (with 13 parties and the SHA-256 circuit, as above). The results are:

	Basic	Red. OT	C-OT	Free-XOR	2 OT-Threads
OT-Off	45631	42408 ± 6300	40644 ± 6266	31071 ± 6720	28805 ± 6878
OT-On		746 ± 56	773 ± 49	664 ± 128	655 ± 81

We also ran our 3-round BGW-based protocol in the LAN setting on the CREATE platform, measuring the influence of optimized multiplications, free-XOR, and multithreading.

	Basic	Opt. Mult.	Free-XOR	2 Mult. Threads
BGW3 Offline	9179 ± 278	8472 ± 294	5809 ± 236	5027 ± 101

4.2 Protocol Comparison – High Latency

Our focus in this paper is the design and implementation of a protocol that is suitable for Internet settings with high latency. We compared our OT-based protocol of Section 3.1 with the BGW-based protocols described in Section 3.2 and with GMW. The running-time of our protocol using multiparty garbled circuits is unaffected by the circuit depth, in contrast to GMW. We therefore ran GMW on different depths for the SHA256-type circuit but not our protocols. The results of these executions are found in Tables 1 and 2.

We remark that the variance of the executions is very large, resulting in some unexplained anomalies. (For example, the online time should be the same for all versions, and should increase with the number of parties. However, in Table 1 you can see that the online time for 7 parties in the OT version is lower than expected.) We conjecture that this is due to the large bandwidth of the protocols and the "noisy" Amazon environment.

		3	7	13
OT	Off	**698 ± 930**	1093 ± 1249	9699 ± 6119
	On	**138 ± 88**	107 ± 87	362 ± 515
BGW3	Off	329 ± 688	2314 ± 1218	9774 ± 8181
	On	143 ± 81	142 ± 76	329 ± 533
BGW2	Off		2212 ± 1440	8745 ± 6832
	On		148 ± 92	264 ± 409
BGW4	Off	498 ± 737	3149 ± 2065	13298 ± 10576
	On	139 ± 78	159 ± 70	308 ± 473
GMW	Off	**231 ± 143**	277 ± 1067	382 ± 290
	On	**3337 ± 166**	3232 ± 9	3341 ± 213

Table 1: The AES circuit – 6800 AND gates (depth 40)

		3	7	13
OT	Off	**6426 ± 1651**	**10291 ± 4968**	**25215 ± 4784**
	On	**172 ± 76**	**226 ± 62**	**456 ± 357**
BGW3	Off	5404 ± 11751	17011 ± 23574	38584 ± 35997
	On	182 ± 77	237 ± 91	520 ± 659
BGW2	Off		14781 ± 12134	37585 ± 17255
	On		283 ± 86	459 ± 325
BGW4	Off	8124 ± 8000	23521 ± 20794	65736 ± 45895
	On	226 ± 78	282 ± 86	454 ± 281
GMW	Off	**850 ± 900**	**5002 ± 10643**	**5042 ± 9212**
(**d=4000**)	On	**309741 ± 32130**	**333996 ± 92024**	**329220 ± 31340**
GMW	Off	701 ± 556	3581 ± 4976	7932 ± 16242
(d=1000)	On	77147 ± 4031	83168 ± 19932	82111 ± 5584
GMW	Off	735 ± 509	2610 ± 8173	4969 ± 9222
(d=100)	On	8038 ± 518	8327 ± 80	8341 ± 271
GMW	Off	598 ± 362	1180 ± 521	5360 ± 12829
(depth=10)	On	880 ± 75	906 ± 25	904 ± 84

Table 2: The SHA256 circuit – 90,825 AND gates

In this Internet-setting, for not shallow circuits, the communication rounds become the dominating factor and so GMW becomes untenable in the online phase for deep circuits. At its most extreme, for the real SHA256 circuit of depth 4000, we have that BMR-OT with 13 parties took 26 seconds overall and under half a second online time, whereas GMW took 335 seconds overall and 330 seconds online time (a difference of one order of magnitude overall, and two orders of magnitude for the online time). This difference grows even larger for 3 parties since GMW is less affected by the number of parties than our protocols.

It is important to note that for low-depth circuits, GMW performs remarkably well even in an Internet setting. This is no surprise for depth-10 and overall time, since the round complexity of both protocols is almost the same. However, even for depth-100, GMW outperforms our protocols for 7 or more parties. This is very informative and is important to explain. The complexity of GMW is linear in the number of parties, versus BMR that is quadratic. Thus, when the number of parties grows, this becomes a factor. Despite this, the *online time* of our protocols are always better (even for depth 10 circuits alone), and the number of parties is *heavily outweighed* by the networking cost for non-shallow circuits. However, for a very large number of parties, the quadratic computation time of BMR-type protocols may become a factor. See Fig. 4 for a graph comparing the total running time.

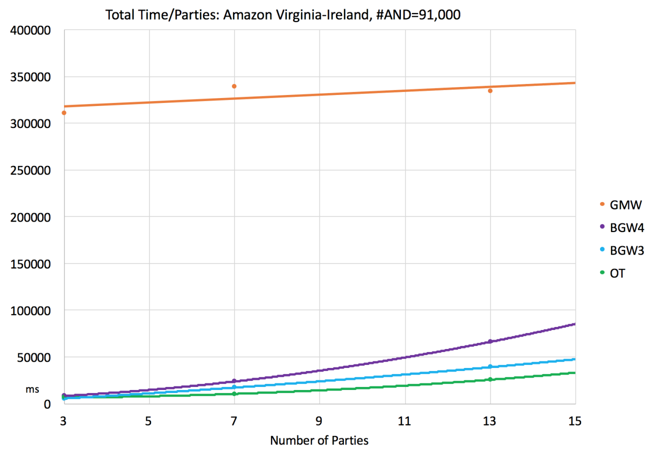

Figure 4: Total time GMW vs BMR in Amazon (Virginia-Ireland)

4.3 Protocol Comparison – Low Latency

Over the past years it has become clear that different secure computation protocols are better suited to different settings. As such, an important question to answer is *when* is one protocol better than another. We therefore ran experiments comparing our protocols on two low latency networks: on the CREATE platform with a ping time of 0.1ms and within the Amazon Virginia region with a ping time of 1ms. Our aim was to understand at what point GMW becomes better than our constant-round protocols based on BMR.

Before proceeding to the results, we review the running-time of the protocol as a function of the number of parties. A theoretical analysis shows that with respect to the number of parties, in the offline phase: GMW scales linearly, the BMR-OT protocol scales *quadratically* (the OT part is linear, but each party sends messages of length $4n\kappa|C|$ to each other party, meaning that each party must send messages of total size $4n^2\kappa|C|$), while the BMR-BGW protocols scale *cubically* (recall that by this we mean that *each* party works linearly, quadratically or cubically). In contrast, in the online phase, the BMR protocols are all the same and we have that GMW scales linearly and BMR scales quadratically.

This analysis yields an interesting and rather complicated tradeoff. On the one hand, GMW scales better than BMR with respect to the number of parties; on the other hand, BMR scales far better as the depth of the circuit increases. This tradeoff becomes more extreme when considering the BGW-based protocols for BMR since they are cubic. As we will see, this is the cause of their downfall, and they rarely perform well.

CREATE – 0.1ms ping time. The aforementioned difference in computational time is most evident in CREATE where latency is very low; see Figure 5 for a graph of the total times including both the offline and online phases (OT stands for the protocol in Section 3.1) Recall that the multiplication optimization for BGW in Section 3.4 only works for up to 15 parties; thus the curve steepens for 33 parties.

Before proceeding, we stress that the scale factor is exactly as expected from the theoretical analysis. However, the big surprise is that in this setting GMW actually outperforms all of the BMR versions of the protocol when the number of parties is 7 or more, even though the circuit has *depth*

4000. In [31], they showed that in the two-party setting, GMW can outperform Yao for low-depth circuits and low-latency networks. Our conclusion here is far more dramatic; in low-latency networks GMW can outperform BMR even for very deep circuits. We stress that BMR protocols have between 3 and 5 rounds whereas GMW has 8000 rounds (2-rounds per level of the circuit). The disparity between this and the two-party setting is due to the significant additional communication and computational overhead in BMR versus Yao (observe that for only a few parties, the running times are comparable).

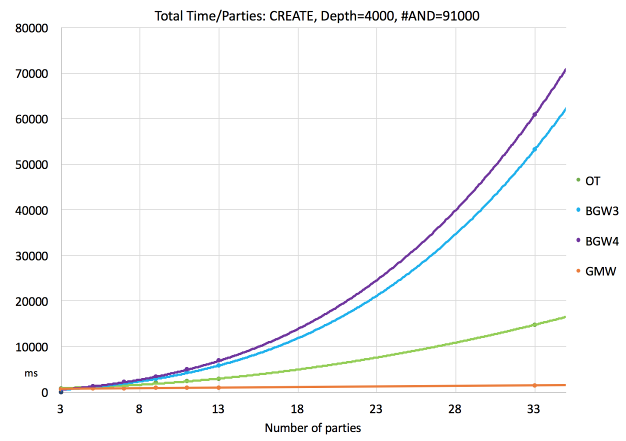

Figure 5: Total time GMW vs BMR in CREATE

Of course, the above considers the total running time. However, the strength of our protocols over GMW is in the *online time*, since BMR requires sending only garbled values on the input wires and *local computation*. In contrast, GMW still has 8000 rounds of communication in the online phase. This means that GMW cannot go below 400ms in the online time. To our great surprise, although BMR was significantly faster than GMW for a small number of parties, *GMW was faster for 33 parties*. Indeed, although for 3 parties, GMW takes approximately 400ms, it grows to only 1113ms for 33 parties. In contrast, BMR takes only 85ms for 3 parties but grows to 1506ms for 33 parties; see Figure 6.

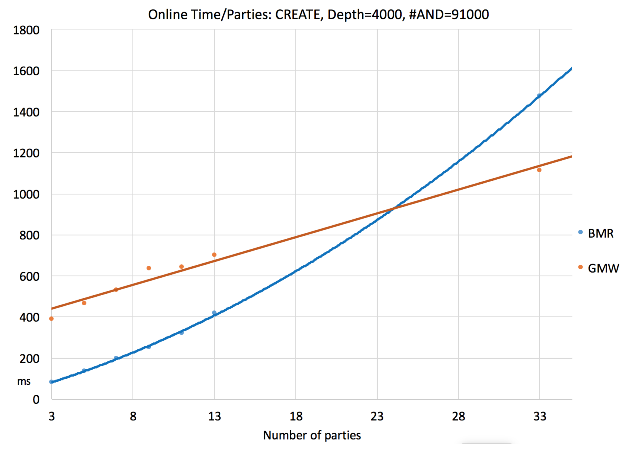

Figure 6: Online time GMW vs BMR in CREATE

We explain this as being due to the quadratic complexity of the BMR circuit. Specifically, evaluating a BMR circuit with 91,000 AND gates for 33 parties requires $33^2 \times 91000 \approx 10{,}000{,}000$ invocations of AES. This therefore becomes a dominating factor. In the two-party setting, the protocol cost has been reduced so significantly that communication is the bottleneck. In the multiparty setting, this is currently *not* the case. (We stress that state-of-the-art pipelining using AES-NI with a fixed-key was used, as in the fastest Yao garbled circuit implementations.)

Amazon Virginia – 1ms ping time. The above relates to a very low latency network with a single message latency of approximately 0.05ms. In the Virginia-Virginia setting of Amazon, a similar phenomenon happens in terms of the scaling curves. However, they are less pronounced because **(a)** the latency of the network plays more of a role, and **(b)** the CPUs on the Amazon machines are stronger, reducing the cost of computation. See Figure 7. For the online time in this setting, the GMW protocol already behaves much more poorly; see Figure 8, even though this is still a fast network (with a 1ms latency). Nevertheless, it is clear from the graph that BMR degrades quickly and so with a larger number of parties (say, 50), GMW will beat BMR even in the online time in Amazon Virginia.

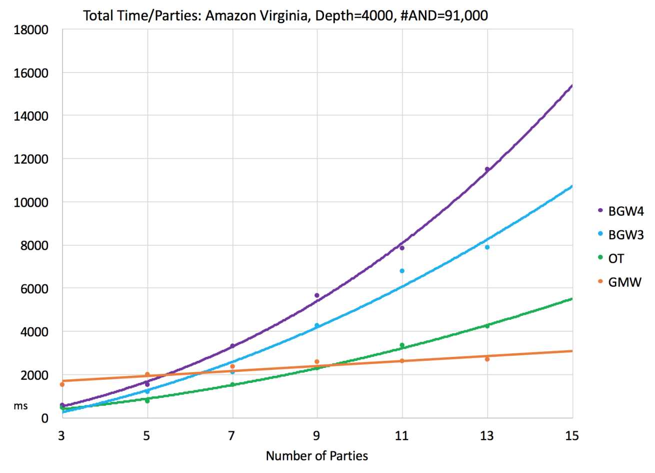

Figure 7: Total time GMW vs BMR in Amazon Virginia

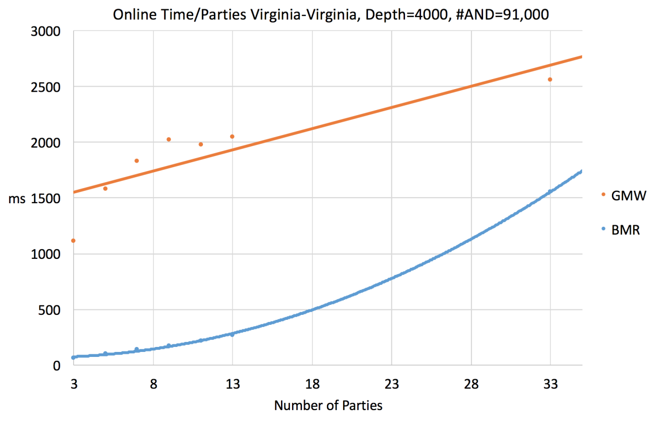

Figure 8: Online time GMW vs BMR in Amazon Virginia

Acknowledgements

We sincerely thank Meital Levy, Assi Barak, Shay Gueron, and Roi Inbar for their help on the implementations and experiments in this paper.

5. REFERENCES

[1] Cyber-defense technology experimental research laboratory (in cooperation with DETER lab). `https://www.create.iucc.ac.il`.

[2] SCAPI – the Secure Computation API. `https://github.com/cryptobiu/libscapi`.

[3] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS*, pages 535–548, 2013.

[4] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In the *28th STOC*, pages 479–488, 1996.

[5] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In the *22nd STOC*, pages 503–513, 1990.

[6] M. Bellare, V.T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Security and Privacy*, pages 478–492, 2013.

[7] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM CCS*, pages 257–266, 2008.

[8] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computations. In the *20th STOC*, pages 1–10, 1988.

[9] D. Bogdanov, S. Laur and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS 2008*, Springer (LNCS 5283), 192–206, 2008.

[10] S.G. Choi, K.W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In *CT-RSA 2012*, pages 416–432, 2012.

[11] S.G. Choi, J. Katz, R. Kumaresan, and H.S. Zhou. On the security of the "free-xor" technique. In the *9th TCC*, pages 39–53, 2012.

[12] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N.P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *18th ESORICS*, pages 1–18, 2013.

[13] I. Damgård, V. Pastro, N.P. Smart and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012*, pages 643–662, 2012.

[14] R. Gennaro, M.O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC'98*, pages 101–111, 1998.

[15] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In the *19th STOC*, 218–229, 1987.

[16] S. Gueron and M.E. Kounavis. Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm. *Inf. Process. Lett.*, 110(14-15):549–553, 2010.

[17] S. Gueron, Y. Lindell, A. Nof and B. Pinkas. Fast Garbling of Circuits Under Standard Assumptions. In the 22*nd ACM CCS*, pages 567–578, 2015.

[18] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In the *USENIX Security Symposium*, 2011.

[19] Y. Ishai and E. Kushilevitz. Randomizing Polynomials: A New Representation with Applications to Round-Efficient Secure Computation. In the 41*st FOCS*, pages 294–304, 2000.

[20] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, pages 145–161, 2003.

[21] Y. Ishai, M. Prabhakaran and A. Sahai. Founding Cryptography on Oblivious Transfer - Efficiently. In *CRYPTO 2008*, pages 572–591, 2008.

[22] M. Keller, E. Orsini and P. Scholl. Actively Secure OT Extension with Optimal Overhead. In *CRYPTO 2015*, Springer (LNCS 9215), pages 724–741, 2015.

[23] M. Keller, P. Scholl and N.P. Smart. An architecture for practical actively secure MPC with dishonest majority. *ACM CCS*, pp. 549–560, 2013.

[24] V. Kolesnikov and R. Kumaresan: Improved OT Extension for Transferring Short Secrets. In *CRYPTO 2013*, Springer (LNCS 8403), pages 54–70, 2013.

[25] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *Automata, Languages and Programming*, pages 486–498, 2008

[26] B. Kreuter, a. shelat, and C. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, pages 285–300, 2012.

[27] E. Larraia, E. Orsini, and N.P. Smart. Dishonest majority multi-party computation for binary circuits. In *CRYPTO 2014*, pages 495–512, 2014.

[28] Y. Lindell, B. Pinkas, N.P. Smart, and A. Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In *CRYPTO 2015*, pages 319–338, 2015.

[29] D. Malkhi, N. Nisan, B. Pinkas and Y. Sella. Fairplay-secure two-party computation system. In the *USENIX Security Symposium*, 2004.

[30] P. Mohassel, M. Rosulek and Y. Zhang. Fast and Secure Three-party Computation: The Garbled Circuit Approach. *ACM CCS*, pp. 591–602, 2015.

[31] T. Schneider and M. Zohner. GMW vs. Yao? efficient secure two-party computation with low depth circuits. In *Financial Cryptography and Data Security*, pages 275–292, 2013.

[32] A. C. Yao. How to generate and exchange secrets. In the *27th FOCS*, pages 162–167, 1986.

[33] S. Zahur, M. Rosulek and D. Evans: Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT*, 220–250, 2015.