Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier

Toshinori Araki*, Assi Barak† Jun Furukawa‡, Tamar Lichter§, Yehuda Lindell†,
Ariel Nof†, Kazuma Ohara*, Adi Watzman¶ and Or Weinstein†

*NEC Corporation, Japan
Email: t-araki@ek.jp.nec.com, k-ohara@ax.jp.nec.com

†Bar-Ilan University, Israel
Email: assaf.barak@biu.ac.il, lindell@biu.ac.il, nofdinar@gmail.com, oror.wn@gmail.com

‡NEC Corporation, Israel
Email: Jun.Furukawa@necam.com

§Queens College, New York, USA; work carried out at Bar-Ilan University
Email: tamar.d.lichter@gmail.com

¶Weizman Institute, Israel; work carried out at Bar-Ilan University
Email: adi.watzman@weizmann.ac.il

***Abstract*—Secure multiparty computation enables a set of parties to securely carry out a joint computation of their private inputs without revealing anything but the output. In the past few years, the efficiency of secure computation protocols has increased in leaps and bounds. However, when considering the case of *security in the presence of malicious adversaries* (who may arbitrarily deviate from the protocol specification), we are still very far from achieving high efficiency. In this paper, we consider the specific case of three parties and an honest majority. We provide general techniques for improving efficiency of cut-and-choose protocols on multiplication triples and utilize them to significantly improve the recently published protocol of Furukawa et al. (ePrint 2016/944). We reduce the bandwidth of their protocol down from 10 bits per AND gate to 7 bits per AND gate, and show how to improve some computationally expensive parts of their protocol. Most notably, we design cache-efficient shuffling techniques for implementing cut-and-choose without randomly permuting large arrays (which is very slow due to continual cache misses). We provide a combinatorial analysis of our techniques, bounding the cheating probability of the adversary. Our implementation achieves a rate of approximately *1.15 billion AND gates* per second on a cluster of three 20-core machines with a 10Gbps network. Thus, we can securely compute 212,000 AES encryptions per second (which is hundreds of times faster than previous work for this setting). Our results demonstrate that high-throughput secure computation for *malicious adversaries* is possible.**

I. Introduction

A. Background

In the setting of secure computation, a set of parties with private inputs wish to compute a joint function of their inputs, without revealing anything but the output. Protocols for secure computation guarantee *privacy* (meaning that the protocol reveals nothing but the output), *correctness* (meaning that the correct function is computed), and more. These security guarantees are provided 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 allowed from the protocol transcript) and *malicious* (where the adversary can run any arbitrary polynomial-time attack strategy). Security in the presence of malicious adversaries provides much stronger security guarantees, but is far more challenging with respect to efficiency.

Feasibility and construction paradigms. Despite its stringent requirements, it has been shown that any polynomial-time functionality can be securely computed with computational security [25], [12], [3] and with information-theoretic security [5], [7]. These results hold both for semi-honest and malicious adversaries, but an honest majority must be assumed in order to obtain information-theoretic security even for semi-honest adversaries. There are two main approaches to constructing secure computation protocols: the secret-sharing approach (followed by [5], [7], [12]) works by having the parties interact for every gate of the circuit, whereas the garbled-circuit approach (followed by [25], [3]) works by having the parties construct an encrypted version of the circuit which can be computed at once. Both approaches have importance and have settings where they perform better than the other. On the one hand, the garbled-circuit approach yields protocols with a constant number of rounds. Thus, in high-latency networks, they far outperform secret-sharing based protocols which have a number of rounds linear in the depth of the circuit being computed. On the other hand, protocols based on secret-sharing have many rounds but can have much lower bandwidth than protocols based on garbled circuits. Thus, in low-latency networks, the secret-sharing approach can potentially achieve a far higher throughput (and reasonable latency for circuits that are not too deep).

As a result, the type of protocol preferred depends very much on whether or not high throughput or low latency is the goal. If low latency is needed (and the circuit being computed is deep), then constant-round protocols like [25] outperform secret-sharing based protocols, even on very fast networks.

DOI 10.1109/SP.2017.15

However, these same protocols fail to achieve high throughput due to the large bandwidth incurred. Due to this situation, it is important to develop protocols for low and high latency networks, with better response time and/or throughput.

In this paper, we focus on the task of achieving *high-throughput* secure computation on low-latency networks, with security for *malicious adversaries*.

Efficiency. The aforementioned feasibility results demonstrate that secure computation is possible in theory, but do not necessarily show how to achieve it efficiently. The problem of constructing efficient protocols for secure computation has gained significant interest recently and progress has been extraordinarily fast, transforming the topic from a notion of theoretical interest only, into a technology that is even being commercialized by multiple companies. In order to demonstrate this progress, it suffices to compare the first implementation in 2004 of a semi-honest two-party protocol based on Yao's garbled circuits that computed at a rate of approximately 620 gates per second [21], to more recent work that processes at a rate of approximately 1.3 million gates per second [13]. 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 [15], [18], [8], [14], [9], [19], [4], [17], [24], [20], [26], [16], [23] for just some examples.

Despite this extraordinary progress, we are still very far away from the goal of achieving high throughput secure computation, especially for malicious adversaries. In 2012, [19] declared the age of "billion-gate secure computation" for malicious adversaries; however, their implementation utilized 256-core machines and took over 2.5 hours to process a billion AND gates, thereby achieving a rate of 100,000 AND gates per second. More recently, two-party secure computation for malicious adversaries was achieved at the rate of over 26,000 AND gates per second on standard hardware [23].

Due to the great difficulty of achieving high throughput for the setting of two parties (and no honest majority in general), we consider the three-party case with an honest-majority. This is interesting in many applications, as described in [22], [1], and is thus worth focusing on.

B. Our Contributions

In this paper, we present a high-throughput protocol for *three-party secure computation with an honest majority and security for malicious adversaries*. We optimize and implement the protocol of [11], that builds on the protocol of [1] that achieves a rate of over 7 billion AND gates per second, but with security only for *semi-honest* adversaries. The multiplication (AND gate) protocol of [1] is very simple; each party sends only a single bit to one other party and needs to compute only a few very simple AND and XOR operations. Security in the presence of malicious adversaries is achieved in [11] by using the cut-and-choose technique in order to generate many valid multiplication triples (shares of *secret* bits (a, b, c) where a, b are random and $c = ab$). These triples are then used to guarantee secure computation, as shown in [2]. This paradigm has been utilized in many protocols; see [20], [9], [16] for just a few examples.

The cut-and-choose method works by first generating many triples, but with the property that a malicious party can make $c \neq ab$. Then, some of the triples are fully "opened" and inspected, to verify that indeed $c = ab$. The rest of the triples are then grouped together in "buckets"; in each bucket, one triple is verified by using all the others in the bucket. This procedure has the property that the verified triple is valid (and a, b, c unknown), unless the unfortunate event occurs that *all* triples in the bucket are invalid. This method is effective since if the adversary causes many triples to be invalid then it is caught when opening triples, and if it makes only a few triples invalid then the chance of a bucket being "fully bad" is very small. The parameters needed (how many triples to open and how many in a bucket) are better – yielding higher efficiency – as the number of triples generated overall increases. Since [1] is so efficient, it is possible to generate a very large number of triples very quickly and thereby obtain a very small bucket size. Using this idea, with a statistical security level of 2^{-40}, the protocol of [11] can generate 2^{20} triples while opening very few and using a bucket size of only 3. In the resulting protocol, each party sends only 10 bits per AND gate, providing the potential of achieving very high throughput.

We carried out a highly-optimized implementation of [11] and obtained a very impressive rate of approximately 500 million AND gates per second. However, our aim is to obtain even higher rates, and the microbenchmarking of our implementation pointed to some significant bottlenecks that must be overcome in order to achieve this. First, in order for cut-and-choose to work, the multiplication triples must be randomly divided into buckets. This requires permuting very large arrays, which turns out to be very expensive computationally due to the large number of cache misses involved (no cache-aware methods for random permutation are known and thus many cache misses occur). In order to understand the effect of this, note that on Intel Haswell chips the L1 cache latency is 4 cycles while the L3 cache latency is 42 cycles [27]. Thus, on a 3.4 GHz processor, the shuffling alone of one billion items in L3 cache would cost 11.7 seconds, making it impossible to achieve a rate of 1 billion gates per second (even using 20 cores). In contrast, in L1 cache the cost would be reduced to just 1.17 seconds, which when spread over 20 cores is not significant. Of course, this is a simplistic and inexact analysis; nevertheless, our experiments confirm this type of behavior.

In addition to addressing this problem, we design protocol variants of the protocol of [11] that require less communications. This is motivated by the assumption that bandwidth is a major factor in the efficiency of the protocol.

Protocol-design contributions. We optimized the protocol of [11], both improving its *theoretical efficiency* (e.g., communication) as well as its *practical efficiency* (e.g., via cache-aware design). We have the following contributions:

1) *Cache-efficient shuffling (Section III-A):* We devise a cache-efficient method of permuting arrays that is sufficient for cut-and-choose. We stress that our method does *not* yield a truly random permutation of the items. Nevertheless, we provide a full combinatorial analysis proving that it suffices for the goal of cut-and-choose. We prove that the probability that an adversary can successfully cheat with our new shuffle technique is the same as when carrying out a truly random permutation.
2) *Reduced bucket size (Section III-B):* As we have described above, in the protocol of [11], each party sends 10 bits to

one other party for every AND gate (when computing 2^{20} AND gates and with a statistical security level of 2^{-40}). This is achieved by taking a bucket size of 3. We reduce the bucket size by 1 and thus the number of multiplication triples that need to be generated and used for verification by *one third*. This saves both communication and computation, and results in a concrete cost of each party sending 7 bits to one other party for every AND gate (instead of 10).

3) *On-demand with smaller buckets (Section III-C):* As will be described below, the improved protocol with smaller buckets works by running an additional shuffle on the array of multiplication triples after the actual circuit multiplications (AND gates) are computed. This is very problematic from a practical standpoint since many computations require far less than 2^{20} AND gates, and reshuffling the entire large array after every small computation is very wasteful. We therefore provide an additional protocol variant that achieves the same efficiency but without this limitation.

All of our protocol improvements and variants involve analyzing different combinatorial games that model what the adversary must do in order to successfully cheat. Since the parameters used in the protocol are crucial to efficiency, we provide (close to) tight analyses of all games.

Implementation contributions. We provide a high-quality implementation of the protocol of [11] and of our protocol variants. By profiling the code, we discovered that the SHA256 hash function computations specified in [11] take a considerable percentage of the computation time. We therefore show how to replace the use of a collision-resistant hash function with a secure MAC and preserve security; surprisingly, this alone resulted in approximately a 15% improvement in throughput. This is described in Section III-D.

We implemented the different protocol variants and ran them on a cluster of three mid-level servers (2.3GHz CPUs with twenty cores) connected by a 10Gbps network. As we describe in Section IV, we used Intel vectorization and a series of optimizations to achieve our results. Due to the practical limitations of the first variant with smaller buckets, we only implemented the on-demand version. The highlights are presented in Table I. Observe that our fastest variant achieves a rate of over ***1.1 billion AND-gates per second***, meaning that large scale secure computation *is* possible even for malicious adversaries.

TABLE I
IMPLEMENTATION RESULTS; THROUGHPUT

Protocol Variant	AND gates/sec	%CPU	Gbps
Baseline [11]	503,766,615	71.7%	4.55
Cache-efficient (SHA256)	765,448,459	64.84%	7.28
Smaller buckets, on-demand (SHA256)	988,216,830	65.8%	6.84
Smaller buckets, on-demand (MAC)	1,152,751,967	71.28%	7.89

Observe that the cache-efficient shuffle alone results in a *50% increase* in throughput, and our best protocol version is *2.3 times faster* than the protocol described in [11].

Offline/online. Our protocols can run in offline/online mode, where multiplication triples are generated in the offline phase and used to validate multiplications in the online phase. The protocol variants with smaller bucket size (items (2) and (3) above) both require additional work in the online phase to randomly match triples to gates. Thus, although these variants have higher throughput, they have a slightly slower online time (providing an interesting tradeoff). We measured the online time only of the fastest online version; this version achieves a processing rate of ***2.1 billion AND gates per second*** (using triples that were previously prepared in the offline phase).

Combinatorial analyses. As we have mentioned above, the combinatorial analyses used to prove the security of our different protocols are crucial for efficiency. Due to this observation, we prove some *independent* claims in Section V that are relevant to all cut-and-choose protocols. First, we ask the question as to whether having different-sized buckets can improve the parameters (intuitively, this is the case since it seems harder for an adversary to fill a bucket with all-bad items if it doesn't know the size of the bucket). We show that this cannot help "much" and it is best to take buckets of all the same size or of two sizes B and $B+1$ for some B. Furthermore, we show that it is possible to somewhat tune the cheating probability of the adversary. Specifically, if a bucket-size B taken does not give a low enough cheating probability then we show that instead of increasing the bucket size to $B+1$ (which is expensive), it is possible to lower the cheating probability moderately at less expense.

C. Related work.

As we have described above, a long series of work has been carried out on making secure computation efficient, both for semi-honest and malicious adversaries. Recent works like [23] provide very good times for the setting of two parties and malicious adversaries (achieving a rate of 26,000 AND gates per second). This is far from the rates we achieve here. However, we stress that they work in a much more difficult setting, where there is no honest majority.

To the best of our knowledge, the only highly-efficient implemented protocol for the case of three parties with an honest majority and (full simulation-based security) for malicious adversaries is that of [22], which follows the garbled-circuit approach. Their protocol achieves a processing rate of approximately 480,000 AND gates per second on a 1Gbps network with single-core machines. One could therefore extrapolate that on a setup like ours, their protocol could achieve rates of approximately 5,000,000 AND gates per second. Note that by [22, Table 3] a single AES circuit of 7200 AND gates requires sending 750KB, or 104 bytes (832 bits) per gate. Thus, on a 10Gbps network their protocol *cannot* process more than 12 million AND gates per second (even assuming 100% utilization of the network, which is typically not possible, and that computation is not a factor). Our protocol is therefore at least *two orders of magnitude* faster. We stress, however, that the *latency* of [22] is much lower than ours, which makes sense given that it follows the garbled circuit approach.

The VIFF framework also considers an honest majority and has an implementation [8]. The offline time alone for preparing 1000 multiplications is approximately 5 seconds. Clearly, on modern hardware, this would be considerably faster, but only by 1-2 orders of magnitude.

II. THE THREE-PARTY PROTOCOL OF [11] – THE BASELINE

A. An Informal Description

In [11], a three-party protocol for securely computing any functionality (represented as a Boolean circuit) with security in

the presence of malicious adversaries and an honest majority was presented. The protocol is extremely efficient; for a statistical cheating probability of 2^{-40} the protocol requires each party to send only 10 bits per AND gate. In this section, we describe the protocol and how it works. Our description is somewhat abstract and omits details about what exact secret sharing method is used, how values are checked and so on. This is due to the fact that all the techniques in this paper are general and work for any instantiation guaranteeing the properties that we describe below. We refer the reader to Appendix A for full details of the protocol of [11].

Background – multiplication triples. The protocol follows the paradigm of generating shares of multiplication triples $([a], [b], [c])$ where $a, b, c \in \{0, 1\}$ such that $c = ab$, and $[x]$ denotes a sharing of x. As we have mentioned, this paradigm was introduced by [2] and has been used extensively to achieve efficient secure computation [20], [9], [16]. These triples have the following properties: it is possible to efficiently validate if a triple is correct (i.e., if $c = ab$) by opening it, and it is possible to efficiently verify if a triple $([a], [b], [c])$ is correct *without opening* it by using another triple $([x], [y], [z])$. This latter check is such that if one triple is correct and the other is not, then the adversary is always caught. Furthermore, nothing is learned about the values a, b, c (but the triple $([x], [y], [z])$ has been "wasted" and cannot be used again).

Protocol description: The protocol of [11] works in the following way:

1) *Step 1 – generate random multiplication triples:* In this step, the parties generate a large number of triples $([a_i], [b_i], [c_i])$ with the guarantee that $[a_i], [b_i]$ are random and all sharings are *valid* (meaning that the secret sharing values held by the honest parties are consistent and of a well-defined value). However, a malicious party can cause $c_i \neq a_i b_i$. In [11] this is achieved in two steps; first generate random sharings of $[a_i], [b_i]$ and then run the semi-honest multiplication protocol of [1] to compute $[c_i]$. This multiplication protocol has the property that the result is *always a valid sharing*, but an adversary can cause $c_i \neq a_i b_i$ and thus it isn't necessarily correct.
2) *Step 2 – validate the multiplication triples:* In this step, the parties validate that the triples generated are valid (meaning that $c_i = a_i b_i$). This is achieved by opening a few of the triples completely to check that they are valid, and to group the rest in "buckets" in which some of the triples are used to validate the others. The validation has the property that all the triples in a bucket are used to validate the first triple, so that if that triple is bad then the adversary is caught cheating unless all the triples in the bucket are bad. The triples are *randomly shuffled* in order to divide them into buckets, and the bucket-size taken so that the probability that there exists a bucket with all-bad triples is negligible. We denote by N the number of triples that need to be generated (i.e., output from this stage), by C the number of triples opened initially, and by B the bucket size. Thus, in order to output N triples in this step, the parties generate $BN + C$ triples in the previous step.
3) *Step 3 – circuit computation:* In this step, the parties securely share their input bits, and then run the semi-honest protocol of [1] up to (but not including) the stage where outputs are revealed. We note that this protocol reveals nothing, and so as long as correctness is preserved, then full security is obtained.
4) *Step 4 – validation of circuit computation:* As we described above, the multiplication protocol used in the circuit computation always yields a valid sharing but not necessarily of the correct result. In this step, each multiplication in the circuit is validated using a multiplication triple generated in Step 2. This uses the exact same procedure of validating "with opening"; as explained above, this reveals nothing about the values used in the circuit multiplication but ensures that the result is correct.
5) *Step 5 – output:* If all of the verifications passed, the parties securely reconstruct the secret sharings on the output wires in order to obtain the output.

The checks of the multiplication triples requires the parties to send values and verify that they have the same view. In order to reduce the bandwidth (which is one of the main aims), in the protocol of [11] the parties compare their views only at the end before the output is revealed, by sending a collision-resistant hash of their view which is very short. (A similar idea of checking only at the end was used in [20], [9]). Note that Steps 1–2 can be run in a separate **offline phase**, reducing latency in the **online phase** of Steps 3–5.

Efficiency. The above protocol can be instantiated very efficiently. For example, sharings of random values can be generated non-interactively, the basic multiplications requires each party sending only a single bit, and verification of correctness of triples can be deferred to the end. Furthermore, since multiplication triples can be generated so efficiently, it is possible to generate a huge amount at once (e.g., 2^{20}) which significantly reduces the overall number of triples required. This is due to the combinatorial analysis of the cut-and-choose game. Concretely, it was shown in [11] that for a cheating probability of 2^{-40}, one can generate $N = 2^{20}$ triples using bucket-size $B = 3$ and opening only $C = 3$ triples. Thus, overall $3N + 3$ triples must be generated. The communication cost of generating each triple initially is a single bit, the cost of each validation (in Steps 2 and 4) is 2 bits, and the cost of multiplying in Step 3 is again 1 bit. Thus, the overall communication per AND gate is just 10 bits per party (3 bits to generate 3 triples, 4 bits to validate the first using the second and third, 1 bit to multiply the actual gate, and 2 bits to validate the multiplication).

Shuffling and generating buckets. The shuffling of Step 2 in [11] works by simply generating a single array of $M = BN + C$ triples and randomly permuting the entire array. Then, the first C triples are opened, and then each bucket is generated by taking B consecutive triples in the array. In our baseline implementation, we modified this process. Specifically, we generate 1 array of length N, and $B-1$ arrays of length $N+C$. The arrays of length $N+C$ are independently shuffled and the last C triples in each of these arrays is opened and checked. Finally, the ith bucket is generated by the taking the ith triple in each of the arrays (for $i = 1, \ldots, N$). This is easier to implement, and will also be needed in our later optimizations. We remark that this is actually a different combinatorial process than the one described and analyzed in [11], and thus must be proven. In Section III-A, we show that this makes almost no difference, and an error of 2^{-40} is

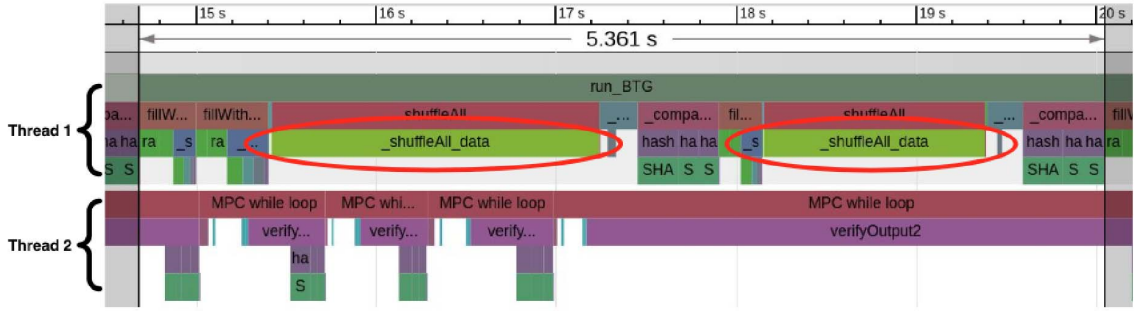

Fig. 1. Microbenchmarking of the baseline implementation (the protocol of [11]), using the CxxProf C++ profiler

achieved when setting $N = 2^{20}$, $B = 3$ and $C = 1$ (practically the same as [11]).

B. Implementation Results and Needed Optimizations

As we have discussed, the above protocol is highly efficient, requiring only 10 bits of communication per AND gate, and requiring only very simple operations. As such, one would expect that a good implementation could achieve a rate that is just a factor of 10 slower than the semi-honest protocol of [1] that computes 7.15 billion AND gates per second. However, our implementation yielded results which fall short of this.

Specifically, on a cluster of three mid-level servers (Intel Xeon E5-2560 v3 2.3GHz with 20 cores) connected by a 10Gbps LAN with a ping time of 0.13 ms, our implementation of [11] achieves a rate of **503,766,615** AND gates per second. This is already very impressive for a protocol achieving malicious security. However, it is **14** times slower than the semi-honest protocol of [1], which is significantly more than the factor of 10 expected by a theoretical analysis.

In order to understand the cause of the inefficiency, see the microbenchmarking results in Figure 1. This is a slice showing one full execution of the protocol, with two threads: the first thread called `run_BTG` (*B*eaver *T*riples *G*enerator) runs Steps 1–2 of the protocol to generate validated triples; these are then used in the second thread called `MPC while loop` to compute and validate the circuit computation (Steps 3–4 of the protocol). Our implementation works on blocks of 256-values at once (using the bit slicing described in [1]), and thus this demonstrates the generation and validation of 256 million triples and secure computation of the AES circuit approximately 47,000 times (utilizing 256 million AND gates).[1]

Observe that *over half the time* in `run_BTG` is spent on just randomly shuffling the arrays in Step 2 (dwarfing all other parts of the protocol). In hindsight, this makes sense since no cache-efficient random shuffle is known, and we use the best known method of Fisher-Yates [10]. Since we shuffle arrays of one million entries of size 256 bits each, this results in either main memory or L3 cache access at almost every swap (since L3 cache is shared between cores, it cannot be utilized when high throughput is targeted via the use of multiple cores). One attempt to solve this is to work with smaller arrays, and so a smaller N. However, in this case, a much larger bucket size will be needed in order to obtain a cheating bound of at most 2^{-40}, significantly harming performance.

Observe also that the fourth execution of `MPC while loop` of the second thread is extremely long. This is due to the fact that `MPC while loop` consumes triples generated by `run_BTG`. In this slice, the first three executions of `MPC while loop` use triples generated in previous executions of `run_BTG`, while the fourth execution of `MPC while loop` is delayed until this `run_BTG` concludes. Thus, the circuit computation thread actually wastes approximately half its time waiting, making the entire system much less efficient.

[1]The *actual times* in the benchmark figure should be ignored since the benchmarking environment is on a local machine and not on the cluster.

III. Protocol Variants and Optimizations

In this section, we present multiple protocol improvements and optimizations to the protocol of [11]. Our variants are all focused on the combinatorial checks of the protocol, and thus do not require new simulation security proofs, but rather new bounds on the cheating probability of the adversary.

Our presentation throughout will assume subprotocols as described in Section II-A: **(a)** generate random multiplication triples, **(b)** verify a triple "with opening", **(c)** verify one triple using another "without opening", and **(d)** verify semi-honest multiplication using a multiplication triple.

A. Cache-Efficient Shuffling for Cut-and-Choose

As we have discussed, one of the major bottlenecks of the protocol of [11] is the cost of random shuffling. In this section, we present a new shuffling process that is *cache efficient*. We stress that our method does not compute a true random permutation over the array. However, it does yield a permutation that is "random enough" for the purpose of cut-and-choose, meaning that the probability that an adversary can obtain a bucket with all bad triples is below the required error.

Informal description. The idea behind our shuffling method is to break the array into subarrays, internally shuffle each subarray separately, and then shuffle the subarrays themselves. By making each subarray small enough to fit into cache (L2 or possibly even L1), and by making the number of subarrays not too large, this yields a much more efficient shuffle. In more detail, recall that as described in Section II-A, instead of shuffling one large array in the baseline protocol, we start with 1 subarray $\vec{D}_1$ of length N, and $B-1$ subarrays $\vec{D}_2, \ldots, \vec{D}_B$ each of size $N+C$, and we shuffle $\vec{D}_2, \ldots, \vec{D}_B$. Our cache-efficient shuffling works by:

1) Splitting each array $\vec{D}_k$ into L subarrays $\vec{D}_{k,1}, \ldots, \vec{D}_{k,L}$.
2) Shuffling each subarray separately. (i.e., randomly permuting the entries inside each $\vec{D}_{k,i}$).
3) Shuffling the subarrays themselves.

This process is depicted in Figure 2.

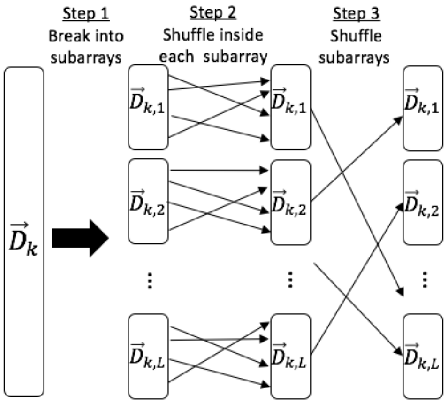

Fig. 2. Cache-efficient shuffling method

We remark that in order to further improve efficiency, we do not shuffle the actual data but rather just the indices.[2] This is much more efficient since it saves many memory copies; we elaborate on this further in Section IV.

As we will show, in order for this to be secure, it is necessary to open C triples in *each subarray*. Thus, $N/L+C$ triples are needed in each subarray, the size of each $\vec{D}_k$ (for $k = 2, \ldots, B$) is $L \cdot (N/L + C) = N + CL$, and the overall number of triples needed is $N+(B-1)(N+CL)$. In addition, overall we execute a shuffling $(B-1)(L+1)$ times: $(B-1)L$ times on the subarrays each of size $N/L+C$ and an additional $B-1$ times on an array of size L. Interestingly, this means that the number of elements shuffled is slightly larger than previously; however, due to the memory efficiency, this is much faster. The formal description appears in Protocol 3.1.

Intuition – security. It is clear that our shuffling process does *not* generate a random permutation over the arrays $\vec{D}_2, \ldots, \vec{D}_B$. However, for cut-and-choose to work, it is seemingly necessary to truly randomly permute the arrays so that the adversary has the lowest probability possible of obtaining a bucket with all-bad triples. Despite this, we formally prove that our method does suffice; we first give some intuition.

Consider the simplistic case that the adversary generates one bad triple in each array $\vec{D}_k$. Then, for every k, the probability that after the shuffling a bad triple in $\vec{D}_k$ will be located in the same index as the bad triple in $\vec{D}_1$ is $\frac{1}{N/L} \cdot \frac{1}{L} = \frac{1}{N}$ (after opening C triples, there are N/L in the subarray and L subarrays; the bad triples will match if they match inside their subarrays and the their subarrays are also matched). Observe that this probability is exactly the same as in the naive shuffling process where the entire array of N is shuffled in entirety.

A subtle issue that arises here is the need to open C triples in *each* of the sub-arrays $\vec{D}_{k,j}$. As we have mentioned, this means that the number of triples that need to be opened increases as L increases. We stress that this is necessary, and it does *not* suffice to open C triples only in the entire array. In order to see why, consider the following adversarial strategy: choose one subarray in *each* $\vec{D}_k$ and make all the triples in the subarray bad. Then, the adversary wins if no bad triple is opened (which happens with probability $1 - \frac{C}{N+C}$) and if the B bad subarray are permuted to the same position (which happens for each with probability $1/L$). The overall probability that the adversary wins is close to $\frac{1}{L^B}$ which is much too large (note that L is typically quite small). By opening balls in each $\vec{D}_{k,j}$, we prevent the adversary from corrupting an entire subarray (or many triples in a subarray).

Before proceeding, note that if we set $L = 1$, we obtain the basic shuffling of Section II-A (and its formal description in Appendix A), and thus the combinatorial analysis provided next, applies to that case as well.

PROTOCOL 3.1 (Generating Valid Triples – Cache-Efficiently):

- **Input:** The number N of triples to be generated.
- **Auxiliary input:** Parameters B, C, X, L, such that $N = (X - C)L$; N is the number of triples to be generated, B is the number of buckets, C the number of triples opened in each subarray, and $X = N/L + C$ is the size of each subarray.
- **The Protocol:**
  1) *Generate random sharings:* The parties generate $2M$ sharings of random values, for $M = 2(N + CL)(B - 1) + 2N$; denote the shares that they receive by $[([a_i], [b_i])]_{i=1}^{M/2}$.
  2) *Generate array $\vec{D}$ of multiplication triples:* As in Step 1 of the informal description in Section II-A (see Protocol A.1).
  3) *Cut and bucket:* In this stage, the parties perform a first verification that the triples were generated correctly, by opening some of the triples.
     a) Each party splits $\vec{D}$ into vectors $\vec{D}_1, \ldots, \vec{D}_B$ such that $\vec{D}_1$ contains N triples and each $\vec{D}_j$ for $j = 2, \ldots, B$ contains $N + LC$ triples.
     b) For $k = 2$ to B: each party splits $\vec{D}_k$ into L subarrays of equal size X, denoted by $\vec{D}_{k,1}, \ldots, \vec{D}_{k,L}$.
     c) For $k = 2, \ldots, B$ and $j = 1, \ldots, L$: the parties jointly and securely generate a random permutation of the vector $\vec{D}_{k,j}$.
     d) For $k = 2, \ldots, B$: the parties jointly and securely generate a random permutation of the vector $[1, \ldots, L]$ and permute the subarrays in $\vec{D}_k$ accordingly.
     e) For $k = 2, \ldots, B$ and $j = 1, \ldots, L$: The parties open and check each of the first C triples in $\vec{D}_{k,j}$, and remove them from $\vec{D}_{k,j}$. If a party rejects any check, it sends $\perp$ to the other parties and outputs $\perp$.
     f) The remaining triples are divided into N sets of triples $\vec{B}_1, \ldots, \vec{B}_N$, each of size B, such that the bucket $\vec{B}_i$ contains the i'th triple in $\vec{D}_1, \ldots, \vec{D}_B$.
  4) *Check buckets:* In each bucket, $B - 1$ triples are used to validate the first (as in Step 2 of the informal description in Section II-A and as in Protocol A.1).
- **Output:** The parties output $\vec{d}$.

Proof of security – combinatorial analysis. We now prove that the adversary can cause the honest parties to output a bad triple in Protocol 3.1 with probability at most $\frac{1}{N^{B-1}}$. This bound is close to tight, and states that it suffices to take $B = 3$ for $N = 2^{20}$ exactly as proven in [11] for the baseline protocol. However, in contrast to the baseline protocol, here the parties must open $(B-1)CL$ triples (instead of just $(B-1)C$). Nevertheless, observe that the bound is actually *independent* of the choice of C and L. Thus, we can take $C = 1$ and we can take L to be whatever is suitable so that $N/L + C$ fits into the cache and L is not too large (if L is large then many triples are wasted in opening and the permutation of $\{1, \ldots, L\}$ would become expensive). Concretely, for $N = 2^{20}$ one could take $L = 512$ and then each subarray is of size 2049 (2048 plus one triple to be

[2]The protocol is highly efficient when using vectorization techniques, as described in Section IV. Thus, each item in the array is actual 256 triples and the data itself is 96 bytes.

opened). Thus, $512(B-1) = 1024$ triples overall are opened when generating 2^{20} triples, which is insignificant.

We start by defining a combinatorial game which is equivalent to the cut-and-bucket protocol using the optimized shuffling process. Recall that C denotes the number of triples that are opened in each subarray, B denotes the size of the bucket, L denotes the number of subarrays, and $X = N/L+C$ denotes the number of triples in each subarray.

$\underline{\text{Game}_1(\mathcal{A}, X, L, B, C)}$:

1) The adversary $\mathcal{A}$ prepares a set D_1 of $(X-C)L$ balls and $B-1$ sets $D_2, \ldots, D_B$ of $X \cdot L$ balls, such that each ball can be either **bad** or **good**.
2) Each set D_k is divided into L subsets $D_{k,1}, \ldots, D_{k,L}$ of size X. Then, for each subset $D_{k,j}$ where $k \in \{2, \ldots, B\}$ and $j \in [L]$, C balls are randomly chosen to be opened. If one of the opened balls is **bad** then output 0. Otherwise, the game proceeds to the next step.
3) Each subset $D_{k,j}$ where $k \in \{2, \ldots, B\}$ and $j \in [L]$ is randomly permuted. Then, for each set D_k where $k \in \{2, \ldots, B\}$, the subsets $D_{k,1}, \ldots, D_{k,L}$ are randomly permuted inside D_k. Denote by $N = L(X-C)$ the size of each set after throwing the balls in the previous step. Then, the balls are divided into N buckets $B_1, \ldots, B_N$, such that B_i contains the ith ball from each set D_k where $k \in [B]$.
4) The output of the game is 1 if and only if there exists i such that bucket B_i is **fully bad**, and all other buckets are either **fully bad** or **fully good**.

We begin by defining the bad-ball profile T_k of a set D_k to be the vector $(t_{k,1}, \ldots, t_{k,L})$ where $t_{k,j}$ denotes the number of bad balls in the j'th subarray of D_k. We say that two sets D_k, D_ℓ have equivalent bad-ball profiles is T_k is a permutation of T_ℓ (i.e., the vectors are comprised of exactly the same values, but possibly in a different order). We begin by proving that the adversary can only win if all sets have equivalent bad-ball profiles.

Lemma 3.2: Let $T_1, \ldots, T_k$ be the bad-ball profiles of $D_1, \ldots, D_L$. If $\text{Game}_1(\mathcal{A}, X, L, B, C) = 1$ then all the bad-ball profiles of $T_1, \ldots, T_k$ are equivalent.

Proof: This is straightforward from the fact the adversary wins (and the output of the game is 1) only if for every $i \in [n]$ all the balls in the ith place of $D_1, \ldots, D_B$ are either bad or good. Formally, assume there exist k, ℓ such that T_k and T_ℓ are not equivalent. Then, for every permutation of the subsets in D_k and D_ℓ, there must exist some j such that $t_{k,j} \neq t_{\ell,j}$ after the permutation. Assume w.l.o.g that $t_{k,j} > t_{\ell,j}$. Then, for every possible permutation of the balls in $D_{k,j}$ and $D_{\ell,j}$, there must be a bad ball in $D_{k,j}$ that is placed in the same bucket as a good ball from $D_{\ell,j}$, and the adversary must lose. Thus, if the adversary wins, then all bad-ball profiles must be equivalent. ■

Next we prove that the best strategy for the adversary is to choose bad balls so that the same number of bad balls appear in every subset containing bad balls. Formally, we say that a bad-ball profile $T = (t_1, \ldots, t_L)$ is single-valued if there exists a value t such for every $i = 1, \ldots, \ell$ it holds that $t_i \in \{0, t\}$ (i.e., every subset has either zero or t bad balls). By Lemma 3.2 we know that all bad-ball profiles must be equivalent in order for the adversary to win. From here on, we can therefore assume that $\mathcal{A}$ works in this way and there is a single bad-ball profile chosen by $\mathcal{A}$. Note that if the adversary chooses no bad balls then it cannot win. Thus, the bad-ball profile chosen by $\mathcal{A}$ must have at least one non-zero value. The following lemma states that the adversary's winning probability is improved by choosing a single-valued bad-ball profile.

Lemma 3.3: Let $T = (t_1, \ldots, t_L)$ be the bad-ball profile chosen by $\mathcal{A}$ and let t be a non-zero value in T. Let $T' = (t'_1, \ldots, t'_L)$ be the bad-ball profile derived from T by setting $t'_i = t$ if $t_i = t$ and setting $t'_i = 0$ otherwise (for every $i = 1, \ldots, L$). Then, $\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \leq \Pr[\text{Game}_1(\mathcal{A}_{T'}, X, L, B, C) = 1]$, where $\mathcal{A}_{T'}$ chooses the balls exactly like $\mathcal{A}$ except that it uses profile T'.

Proof: Let T be the bad-ball profile chosen by $\mathcal{A}$ and define T' as in the lemma. Let E_1 denote the event that no bad balls were detected when opening C balls in every subset, that all subsets containing t bad balls are matched together, and that all bad balls in these subsets containing t bad balls are matched in the same buckets. By the definition of the game, it follows that $\Pr[\text{Game}_1(\mathcal{A}_{T'}, X, L, B, C) = 1] = \Pr[E_1]$. Next, define by E_2 the probability that in the game with $\mathcal{A}$, the subsets with a number of bad balls not equal to t are matched and bucketed together. Then,

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] = \Pr[E_1 \wedge E_2].$$

We have that

$$\begin{aligned} &\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\ &= \Pr[E_1 \wedge E_2] = \Pr[E_2 \mid E_1] \cdot \Pr[E_1] \\ &\leq \Pr[E_1] = \Pr[\text{Game}_1(\mathcal{A}_{T'}, X, L, B, C) = 1] \end{aligned}$$

and the lemma holds. ■

We are now ready to prove that the adversary can win in the game with probability at most $1/N^{B-1}$ (independently of C, N, as long as $C > 0$).

Theorem 3.4: For every adversary $\mathcal{A}$, for every $L > 0$ and $0 < C < X$, it holds that

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \leq \frac{1}{N^{B-1}}$$

where $N = (X-C)L$.

Proof: By Lemma 3.3 it follows that the best strategy for the adversary is to choose some S subsets from each set, and to put exactly t bad balls in each of them, for some t, while all other subsets contain only good balls. Thus, overall there are SB subsets containing bad balls.

Next, we analyze the success probability of the adversary in the game when using this strategy. We define three *independent* events:

$\boldsymbol{E_c}$: the event that no bad balls were detected when opening C balls in each of the S sub-sets containing t bad balls in $D_2, \ldots, D_B$. Since there are $\binom{X}{C}$ ways to choose C balls out of X balls, and $\binom{X-t}{C}$ ways to choose C balls without choosing any of the t bad balls, we obtain that

$$\Pr[E_c] = \left(\frac{\binom{X-t}{C}}{\binom{X}{C}}\right)^{S(B-1)} = \left(\frac{(X-t)!(X-C)!}{X!(X-t-C)!}\right)^{S(B-1)} \quad (1)$$

$\boldsymbol{E_L}$: the event that after permuting the subsets in $D_2, \ldots, D_B$, the S subsets containing t bad balls are positioned at the same locations of the S subsets in D_1. There are $L!$ ways to permute the subsets in each D_k, and $S!(L-S)!$ ways to permute such

that the subsets with t bad balls will be in the same location as in D_1. Thus, we have

$$\Pr[E_L] = \left(\frac{S!(L-S)!}{L!}\right)^{B-1} = \binom{L}{S}^{-(B-1)}$$

$\boldsymbol{E_t}$: the event that after permuting the balls inside the subsets, all bad balls are positioned in the same location in $D_1, \ldots, D_B$. For subset $D_{j,k}$ which contains t bad balls, there are $(X-C)!$ ways to permute it. In contrast, there are only $t!(X-C-t)!$ ways to permute it such that the bad balls will be in the same location of the bad balls in $D_{1,k}$. Since there are S subsets with t bad balls in each set, we have that

$$\Pr[E_t] = \left(\frac{t!(X-C-t)!}{(X-C)!}\right)^{S(B-1)}. \quad (2)$$

Combining the above three equations and noting that the product of Eq. (1) and Eq. (2) equals $\left(\frac{t!(X-t)!}{X!}\right)^{S(B-1)} = \binom{X}{t}^{-S(B-1)}$, we conclude that

$$\begin{aligned}
&\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\
&= \Pr[E_c \wedge E_L \wedge E_t] = \Pr[E_c] \cdot \Pr[E_L] \cdot \Pr[E_t] \\
&= \binom{L}{S}^{-(B-1)} \binom{X}{t}^{-S(B-1)}. \quad (3)
\end{aligned}$$

Next, observe that for the adversary to win it must hold that $t \leq X - C < X$ and $S > 0$. Thus, we can use the fact that for every $0 < t < X$ it holds that $\binom{X}{t} \geq \binom{X}{1}$. In contrast, the adversary may choose to corrupt all subarrays. i.e., set $S = L$. Thus, we consider two cases.

- *Case 1*: $S = L$. In this case, we obtain that

$$\begin{aligned}
&\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\
&= \binom{L}{L}^{-(B-1)} \binom{X}{t}^{-L(B-1)} \\
&\leq \binom{X}{1}^{-L(B-1)} = \frac{1}{X^{L(B-1)}}.
\end{aligned}$$

- *Case 2*: $0 < S < L$. In this case, we obtain that

$$\begin{aligned}
&\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\
&\leq \binom{L}{1}^{-(B-1)} \binom{X}{1}^{-S(B-1)} \\
&= \frac{1}{(L \cdot X^S)^{B-1}} \leq \frac{1}{(L \cdot X)^{B-1}}
\end{aligned}$$

Since for every $L > 0$ and $X > 1$ (as assumed in the theorem) it holds that $L \cdot X \leq X^L$, we conclude that

$$\begin{aligned}
&\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\
&\leq \max\left(\frac{1}{X^{L(B-1)}}, \frac{1}{(L \cdot X)^{B-1}}\right) = \frac{1}{(L \cdot X)^{B-1}} \\
&< \frac{1}{\Big(L(X-C)\Big)^{B-1}} = \frac{1}{N^{B-1}}.
\end{aligned}$$

By setting $\frac{1}{N^{B-1}} \leq 2^{-\sigma}$ in Theorem 3.4, we conclude: ∎

Corollary 3.5: If L, X, C and B are chosen such that $\sigma \leq (B-1)\log N$ where $L > 0$, $X > C > 0$ and $N = (X-C)L$, then for every adversary $\mathcal{A}$, it holds that $\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \leq 2^{-\sigma}$.

Concrete parameters. Observe that for $N = 2^{20}$, it suffices to set $B = 3$ and $C = 1$ and for any L we have that the adversary wins with probability at most 2^{-40}. This thus achieves the tight analysis provided in [11] when a cache-*inefficient* shuffle is used. In our implementation, we take $N = 2^{20}$ and $L = 2^9$; thus we have 512 subarrays of size 2048 each. Recall that we actually only shuffle the indices; for a subarray of length 2048 we need indices of size 2 bytes and so the entire subarray to be shuffled is 4096 bytes = 4KB. This fits into the L1 cache on most processors making the shuffle very fast.

B. Reducing Bucket-Size and Communication

Clearly, the major cost of the protocol is in generating, shuffling and checking the triples. If it were possible to reduce the size of the buckets needed, this would in turn reduce the number of triples to be generated and result in a considerable saving. In particular, the protocol of [11] uses a bucket size of 3 and requires that each party send 10 bits per AND gate; this places a strict lower bound on performance dependent on the available bandwidth. In this section, we show how to reduce the bucket size by 1 (concretely from 3 to 2) and thereby reduce the number of triples generated by 1/3, reducing computation and communication. Formally, we present an improvement that reduces the cheating probability of the adversary from $\frac{1}{N^{B-1}}$ to $\frac{1}{N^B}$, thus enabling us to use $B' = B-1$. Thus, if previously we needed to generate approximately 3 million triples in order to compute 1 million AND gates, in this section we show how the same level of security can be achieved using only *2 million* triples. Overall, this reduces communication from 10 bits per AND gate to 7 bits per AND gate (since 1 bit is needed to generate a triple and 2 bits are needed to verify each triple using another).

The idea behind the protocol improvement is as follows. The verification of a multiplication gate in the circuit uses one multiplication triple, with the property that if the gate is incorrect and the triple is valid (meaning that $c = ab$), then the adversary will be caught with probability 1. Thus, as long as all triples are valid with very high probability, the adversary cannot cheat. The improvement that we propose here works by observing that if a correct multiplication gate is verified using an incorrect triple *or* an incorrect multiplication gate is verified using a correct triple, then the adversary will be caught. Thus, if the array of multiplication triples is randomly shuffled *after* the circuit is computed, then the adversary can only successfully cheat if the random shuffle happens to match good triples with good gates and bad triples with bad gates. As we will see, this significantly reduces the probability that the adversary can cheat and so the bucket size can be reduced by 1. Note that although the number of triples is reduced (since the bucket size is reduced), the number of shuffles remains the same. The modified protocol is described in Protocol 3.6; the steps reference the formal specification of [11] in Appendix A.

Observe that in the reshuffle stage in Protocol 3.6, the random permutation is computed over the *entire* array, and does not use the cache-efficient shuffling of Section III-A. This is due to the fact that unlike the triples generated in the preprocessing/offline phase, no triples of the circuit emulation phase can be opened. Thus, the adversary could actually make as many triples as it wishes in the circuit emulation phase be incorrect. In order to see why this is a problem, consider

PROTOCOL 3.6 (Secure Computation – Smaller Buckets):

- **Inputs and Auxiliary Input:** Same as in Protocol A.2; In addition, the parties hold a parameter L.
- **The protocol – offline phase:** Generate multiplication triples by calling Protocol 3.1; let $\vec{d}$ be the output.
- **The protocol – online phase:**
  1) *Input sharing and circuit emulation:* Exactly as in Protocol A.2.
  2) *Reshuffle stage:* The parties jointly and securely generate a random permutation over $\{1,\ldots,N\}$ and then each locally shuffle $\vec{d}$ accordingly.
  3) *Verification and output stages:* Exactly as in Protocol A.2.

the case that the adversary choose to corrupt one subarray in each of $\vec{D}_1,\ldots,\vec{D}_B$ and make $X-1$ out of the X triples incorrect. Since $C=1$, this implies that the adversary is *not caught* when opening triples in subarrays $\vec{D}_2,\ldots,\vec{D}_B$ with probability X^{-B+1}. Furthermore, the probability that these subarrays with all-bad triples are matched equals L^{-B+1}. Thus, the adversary succeeds in have an all-bad bucket in the preprocessing phase with probability $\frac{1}{(XL)^{B-1}} < \frac{1}{N^{B-1}}$ as proven in Theorem 3.4. Now, if the cache-efficient shuffle is further used in the circuit computation phase, then the adversary can make a subarray all-bad there as well (recall that nothing is opened) and this will be matched with probability $1/L$ only. Thus, the overall cheating probability is bounded by $\frac{1}{L}\cdot\frac{1}{N^{B-1}} >> \frac{1}{N^B}$. As a result, the shuffling procedure used in the online circuit-computation phase is a full permutation, and not the cache-efficient method of Section III-A. We remark that even when using a full permutation shuffle, we need to make an additional assumption regarding the parameters. However, this assumption is fortunately very mild and easy to meet, as will be apparent below.

As previously, we begin by defining a combinatorial game to model this protocol variant.

$\underline{\text{Game}_2(\mathcal{A},X,L,B,C)}$:

1) Run $\text{Game}_1(\mathcal{A},X,L,B,C)$ once. If the output is 0, then output 0. Otherwise, proceed to the next step with the buckets $B_1,\ldots B_N$.
2) The adversary $\mathcal{A}$ prepares an additional set $\vec{d}$ of N balls where each ball can be either **bad** or **good**.
3) The set $\vec{d}$ is shuffled. Then, the ith ball in $\vec{d}$ is added to the bucket B_i.
4) The output of the game is 1 if and only if each bucket is **fully good** or **fully bad**.

Note that in this game we do not explicitly require that the adversary can only win if there exists a bucket that is fully bad, since this condition is already fulfilled by the execution of Game_1 in the first step. We proceed to bound the winning probability of the adversary in this game.

Theorem 3.7: If $B\geq 2$, then for every adversary $\mathcal{A}$ and for every $L>0$ and $0<C<X$ such that $X^L \geq (X\cdot L)^2$, it holds that

$$\Pr[\text{Game}_2(\mathcal{A},X,L,B,C)=1] \leq \frac{1}{N^B}$$

where $N=(X-C)L$.

Proof: Assume that the adversary chooses to corrupt exactly S subsets in Game_1 (the first step of Game_2) by inserts exactly t bad balls in each (recall that this strategy is always better, as proven in Lemma 3.3). Then, as shown in Eq. (3) in the proof of Theorem 3.4, it holds that

$$\Pr[\text{Game}_1(\mathcal{A},X,L,B,C)=1]=\binom{L}{S}^{-(B-1)}\binom{X}{t}^{-S(B-1)}.$$

Next, it is easy to see that for the adversary to win in Game_2, it must choose exactly $S\cdot t$ bad balls in $\vec{d}$ (otherwise a good and bad ball with certainly be in the same bucket). There are $\binom{N}{S\cdot t}$ ways of matching the $S\cdot t$ bad balls in $\vec{d}$, and there is exactly one way in which the adversary wins (this is where all $S\cdot t$ match the bad balls from Game_1). Thus, the probability that the adversary wins is

$$\begin{aligned}&\Pr[\text{Game}_2(\mathcal{A},X,L,B,C)=1]\\ &= \binom{N}{S\cdot t}^{-1}\cdot\Pr[\text{Game}_1(\mathcal{A},X,L,B,C)=1]. \quad (4)\end{aligned}$$

We separately consider two cases:

- *Case 1 – $S\cdot t<N$:* Applying Eq. (4) and the fact that $\binom{N}{S\cdot t}^{-1}$ is maximized for $S\cdot t=1$ (since $S\cdot t$ cannot equal 0 or N)
$$\Pr[\text{Game}_2(\mathcal{A},X,L,B,C)=1] \leq \binom{N}{1}^{-1}\cdot\frac{1}{N^{B-1}}=\frac{1}{N^B}$$
- *Case 2 – $S\cdot t=N$:* Observe that we cannot use Eq. (4) here since $\binom{N}{N}=1$. We therefore prove the bound using Eq. (3). Here $S=L$ and so $\binom{L}{S}=1$ and $t=X-C$ (note that $t<X$ since $C>0$). Plugging this into Eq. (3) we have
$$\begin{aligned}&\Pr[\text{Game}_2(\mathcal{A},X,L,B,C)=1]\\ &= \binom{X}{X-C}^{-L(B-1)} \leq \binom{X}{1}^{-L(B-1)} = \frac{1}{X^{L(B-1)}}.\end{aligned}$$
Now, using the assumption that $X^L\geq(X\cdot L)^2$, which implies $X^{L(B-1)}\geq(X\cdot L)^{2(B-1)}\geq(X\cdot L)^B$ when $B\geq 2$ (which is indeed the minimal size of a bucket as assumed in the theorem), we obtain that
$$\begin{aligned}&\Pr[\text{Game}_2(\mathcal{A},X,L,B,C)=1]\\ &= \frac{1}{X^{L(B-1)}}\leq\frac{1}{(L\cdot X)^B}\leq\frac{1}{(L\cdot(X-C))^B}=\frac{1}{N^B}.\end{aligned}$$
We have the following corollary: ∎

Corollary 3.8: Let L,X,C and B be such that $\sigma\leq B\log N$ where $B\geq 2$, $L<0$, $0<C<X>0$, $X^L\geq(X\cdot L)^2$ and $N=(X-C)L$. Then for every adversary $\mathcal{A}$, it holds that $\Pr[\text{Game}_2(\mathcal{A},X,L,B,C)=1]\leq 2^{-\sigma}$

Concrete parameters and a tradeoff. As we have described above, this shows that setting $C=1$, $B=2$ and X,L such that $N=(X-C)L=2^{20}$ yields a security bound of 2^{-40} as desired. Thus, we can reduce the size of each bucket by 1, and can use only 2 arrays in the triple generation phase (shuffling just one of them), at the expense of an additional shuffle in the online phase.

Clearly, in some cases one would not settle on any increase of the online work. Nevertheless, our analysis gives a clear trade-off of the offline communication complexity vs. the online computational complexity.

The latency vs throughput tradeoff. By reducing the number of triples sent and by reducing the communication, the protocol improvement here should considerably improve

throughput. However, it is important to note that the fact that the online shuffle is not cache efficient means that the throughput increase is not optimal. In addition, it also means that the *online time* is considerably increased. Thus, when the secure computation is used in an application where low latency is needed, then this improvement may not be suitable. However, when the focus is on high throughput secure computation, this is of importance.

Practical limitations. Although theoretically attractive, in most practical settings, the implementation of this protocol improvement is actually very problematic. Specifically, if a circuit computation involving N AND gates is used for a large N, then the improvement is suitable. However, in many (if not most) cases, circuits of smaller sizes are used and a large N is desired in order to achieve good parameters. For example, 2^{20} triples suffice for approximately 180 AES computation. In such a case, this protocol variant cannot be used. In particular, either the application has to wait for all AES computations to complete before beginning verification of the first (recall that the shuffle must take place *after* the circuit computation) or a full shuffle of what is left of the large array must be carried out after each computation. The former solution is completely unreasonable for most applications and the latter solution will result in a very significant performance penalty. We address this issue in the next section.

C. Smaller Buckets With On-Demand Secure Computation

In this section, we address the problem described at the end of Section III-B. Specifically, we describe a protocol variant that has smaller buckets as in Section III-B, but enables the utilization of multiplication triples *on demand* without reshuffling large arrays multiple times. Thus, this protocol variant is suitable for settings where many triples are generated and then used on-demand as requests for secure computations are received by an application.

The protocol variant that we present here, described in Protocol 3.9, works in the following way. First, we generate 2 arrays $\vec{d_1}, \vec{d_2}$ of N multiplication triples each, using Protocol 3.1 (and using a smaller B as in Section III-B). Then, in order to verify a multiplication gate, a random triple is chosen from $\vec{d_1}$ and replaced with the next unused triple in $\vec{d_2}$. After N multiplication gates have been processed, the triples in $\vec{d_2}$ will be all used and Protocol 3.1 will be called again to replenish it. Note that $\vec{d_1}$ always contains N triples, as any used triple is immediately replaced using $\vec{d_2}$.

As before, we need to show that this way of working achieves the same level of security as when a full shuffle is run on the array. Formally, we will show that for N triples and buckets of size B, the probability that the adversary succeeds in cheating is bounded by $\frac{1}{N^B}$, just as in Section III-B. Note that Protocol 3.9 as described is actually *continuous* and does not halt. Nevertheless, for simplicity, we start by presenting the bound for the case of computing N gates, and then use it for computing the bound in the continuous analysis.

In order to prove the bound, we begin by defining the combinatorial game $\text{Game}_3(\mathcal{A}, X, L, B, C)$ which is equivalent to the process described in Protocol 3.9 When computing N gates.

PROTOCOL 3.9 (On-Demand with Smaller Buckets):

- **Inputs and Auxiliary Input:** Same as in Protocol A.2.
- **The protocol – triple initialization:**
  1) The parties run Protocol 3.1 twice with input N and obtain two vectors $\vec{d_1}$,$\vec{d_2}$ of sharings of random multiplication triples.
- **The protocol – circuit computation:** Upon receiving a request to compute a circuit:
  1) *Sharing the inputs:* Same as in Protocol A.2.
  2) *Circuit emulation:* Same as in Protocol A.2.
  3) *Verification stage: Before* the secrets on the output wires are reconstructed, the parties verify that all the multiplications were carried out correctly, as follows. For $k = 1, \ldots, N$:
     a) Denote by $([x], [y])$ the shares of the input wires to the kth AND gate, and denote by $[z]$ the shares of the output wire of the kth AND gate.
     b) The parties run a secure coin-tossing protocol in order to generate a random $j \in [N]$. (In [11], it is shown that secure coin-tossing can be non-interactively and efficiently computed in this setting.)
     c) The parties check the triple $([x], [y], [z])$ using $([a_j], [b_j], [c_j])$ (the jth triple in $\vec{d_1}$).
     d) If a party rejects any of the checks, it sends $\perp$ to the other parties and outputs $\perp$.
     e) Each party replaces its shares of $([a_j], [b_j], [c_j])$ in $\vec{d_1}$ with the next unused triple in $\vec{d_2}$.
  4) *Output reconstruction and output:* Same as in Protocol A.2.
- **Replenish:** If $\vec{d_2}$ is empty (or close to empty) then the parties run Protocol 3.1 with input N to obtain a new $\vec{d_2}$.

$\underline{\text{Game}_3(\mathcal{A}, X, L, B, C)}$:

1) Run steps 1-3 of $\text{Game}_1(\mathcal{A}, X, L, B, C)$ twice to receive two lists of buckets $B_1, \ldots, B_N$ and $B'_1, \ldots, B'_N$.
2) If all buckets are either **fully good** or **fully bad** proceed to the next step. Otherwise, output 0.
3) The adversary $\mathcal{A}$ prepares N new balls denoted by $b_1, \ldots, b_N$, where each ball can be either **bad** or **good**, with the requirement that at least one of the balls must be bad.
4) For $i = 1$ to N:
   a) The ball b_i is thrown into a random bucket B_k ($k \in [N]$).
   b) If the bucket B_k is **fully bad** output 1.
   c) If the bucket B_k is not **fully good** or **fully bad** output 0.
   d) Replace B_k with the bucket B'_i.

Observe that in this game, the adversary is forced to choose a bad ball only when it prepares the N additional balls. This means that in order for it to win, there must be at least one bad bucket among $B_1, \ldots, B_N$. For this to happen, the adversary must win in at least one of Game_1 executions. Thus, in the proof of the following theorem, we will use the bound stating that the probability that the adversary wins in Game_1 is at most $1/N^{B-1}$. In addition, note that from the condition in the last step, the adversary wins if and only if the *first bad ball* is thrown into a fully bad bucket (even if a bad ball is later thrown into a fully good bucket meaning that the adversary will be detected). This is in contrast to previous games where the adversary only wins if *all* bad balls are thrown into fully bad buckets. This is due to the fact that output may be provided after only using some of the triples. If one of the triples was bad, then this will be a breach of security, and the fact that the adversary is caught later does not help (in the sense that security was already broken). Thus, cheating must be detected at the first bad ball and no later.

We consider the case of $B \geq 2$ and our aim is to prove that the probability that the adversary cheats is at most $1/N^B$. In this game, unlike Sections III-A and III-B, we actually need to open at least $C = 3$ triples in each subarray. We will explain why this is necessary at the end of the proof.

We prove the theorem under the assumptions that $X > L + C$ (meaning that the number of subarrays is less than the size of each subarray), that $L \geq 5$ (meaning that there are at least 5 subarrays), that $C \geq 3$, and that $X - C \geq 6$ (meaning that the subarrays are at least of size $C+6$ which can equal 9). All of these conditions are fulfilled for reasonable choices of parameters in practice.

Theorem 3.10: Let $B \geq 2$ and assume $X > L+C$. Then for every adversary $\mathcal{A}$ and for every $L \geq 5$, $C \geq 3$ and $X-C \geq 6$ it holds that

$$\Pr[\text{Game}_3(\mathcal{A}, X, L, B, C) = 1] \leq \frac{1}{N^B}$$

where $N = (X - C)L$.

Proof: In order to win the game, $\mathcal{A}$ must choose bad balls in at least one of Game$_1$ executions. If $\mathcal{A}$ chooses bad balls in both executions, then the theorem follows directly from Theorem 3.4, since $\mathcal{A}$ wins in two executions of Game$_1$ with probability only $\frac{1}{N^{B-1}} \cdot \frac{1}{N^{B-1}} = \frac{1}{N^{2B-2}} \leq \frac{1}{N^B}$, where the last inequality holds when $B \geq 2$ as assumed, in the theorem.

Thus, for the remainder of the proof we assume that $\mathcal{A}$ chose bad balls in exactly one of Game$_1$ executions only (note that the cases are mutually exclusive and so the probability of winning is the maximum probability of both cases).

Denote by S the number of subsets that contain bad buckets after the Game$_1$ executions (recall that we consider the case only that these are all in the same Game$_1$ execution), and let t be the number of bad buckets (in the proof of Theorem 3.4, note that t denotes the number of bad balls in the subarray; if the adversary is not caught then this is equivalent to the number of bad buckets). By Eq. (3) we have that

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] = \binom{L}{S}^{-(B-1)} \binom{X}{t}^{-S(B-1)}.$$

We separately consider the cases that $S = 1$, $S = 2$, $S = 3$ and $S \geq 4$.

Case 1 – $S = 1$: In this case, we have

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] = \binom{L}{1}^{-(B-1)} \binom{X}{t}^{-(B-1)} = \left(L \cdot \binom{X}{t}\right)^{-(B-1)}.$$

If $t = 1$, then $\binom{X}{t}^{-1} = \frac{1}{X} < \frac{L}{N}$ and so the probability that $\mathcal{A}$ wins in Game$_1$ is at most $\frac{1}{N^{B-1}}$. In this case, in the latter steps in Game$_3$, $\mathcal{A}$ can only win by choosing exactly one bad ball out of $b_1, \ldots, b_N$. Now, this ball is thrown into a random bucket, and there is at most one bad bucket (note that if the bad bucket is in the second array then depending on where the bad ball is, it may not even be possible for it to be chosen). Thus, the probability that it will be thrown into that bucket (which is essential for $\mathcal{A}$ to win) is at most $\frac{1}{N}$. Overall, we have that $\mathcal{A}$ can win Game$_3$ with probability at most $\frac{1}{N^B}$ (since $\mathcal{A}$ must both win in Game$_1$ and have the bad ball thrown in the single bad bucket).

Next, if $t = 2$, we have that

$$\binom{X}{t}^{-(B-1)} = \binom{X}{2}^{-(B-1)} = \left(\frac{2}{X(X-1)}\right)^{B-1}.$$

Thus,

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] = \left(\frac{2}{L \cdot X(X-1)}\right)^{B-1}.$$

Now, in the later phase of Game$_3$, we have that there are at most 2 bad buckets out of N buckets overall.[3] Thus, for each bad ball, the probability that it will be thrown into a bad bucket is at most $\frac{2}{N}$. Combining these together, we have that the adversary can win in Game$_3$ with probability at most

$$\frac{2}{N} \cdot \left(\frac{2}{L \cdot X(X-1)}\right)^{B-1} = \frac{2^B}{L^B X^{B-1}(X-1)^{B-1}(X-C)} < \frac{2^B}{L^B X^{B-1}(X-C)^B}$$

and since we assume $X \geq 4$ and thus $2^B \leq X^{B-1}$ for $B \geq 2$, the above is at most

$$\frac{1}{L^B(X-C)^B} = \frac{1}{N^B}.$$

Finally, if $t \geq 3$, we have that

$$\binom{X}{t}^{-1} \leq \binom{X}{3}^{-1} = \frac{6}{X(X-1)(X-2)} < \frac{6}{(X-C)^3} = \frac{6L^3}{N^3}$$

and hence

$$\binom{X}{t}^{-(B-1)} < \left(\frac{6L^3}{N^3}\right)^{B-1}.$$

We stress that $\binom{X}{t}^{-1} \leq \binom{X}{3}^{-1}$ is only true since we take $C \geq 3$, because otherwise $\binom{X}{t}^{-1}$ would be smaller for $t = X - 2$ or $t = X - 1$. However, when three balls are checked, if the adversary sets $t \geq X - 2$ it will certainly be caught (since at least one bad ball will always be checked). Thus,

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] < \left(\frac{6L^2}{N^3}\right)^{B-1}.$$

Now, in the later phase of Game$_3$, we have that there are at most $\frac{N}{L}$ bad buckets out of N buckets overall (since $S = 1$). Thus, for each bad ball, the probability that it will be thrown into a bad bucket is at most $\frac{1}{L}$. Combining these together, we

[3]This holds since $t = 2$ and thus 2 bad buckets were generated in Game$_1$. Note that there are *at most* 2 bad buckets at this stage and not necessarily 2 since the bad buckets in Game$_1$ may have been generated in the second set.

have that the adversary can win in Game_3 with probability at most

$$\begin{aligned}\frac{1}{L}\cdot\left(\frac{6L^2}{N^3}\right)^{B-1} &= \frac{6^{B-1}L^{2(B-1)}}{L^{3(B-1)}(X-C)^{3(B-1)}L}\\ &= \frac{6^{B-1}}{L^B(X-C)^{3(B-1)}}\\ &= \frac{6^{B-1}}{N^B(X-C)^{2(B-1)}}\\ &< \frac{1}{N^B}\end{aligned}$$

where the inequality follows since we assume $X-C\geq 6$ and $B\geq 2$, and thus $6^{B-1}<(X-C)^{2(B-1)}$.

Case 2 – $S=2$: Observing that $\binom{L}{2}=\frac{L(L-1)}{2}$ and recalling that $\binom{X}{t}^{-1}\leq\binom{X}{1}^{-1}=\frac{1}{X}<\frac{L}{N}$, in this case we have

$$\begin{aligned}&\Pr[\text{Game}_1(\mathcal{A},X,L,B,C)=1]\\ &< \left(\frac{2}{L(L-1)}\right)^{B-1}\left(\frac{L}{N}\right)^{2(B-1)}\\ &= \left(\frac{2L^2}{L(L-1)N^2}\right)^{B-1}\\ &= \left(\frac{2L}{(L-1)N^2}\right)^{B-1}.\end{aligned}$$

Now, since $S=2$ we have that at most 2 subarrays were corrupted and so the number of bad buckets from Game_1 is at most $2\cdot\frac{N}{L}$. Thus, the probability that a bad ball is thrown into a bad bucket is at most $\frac{2}{L}$, and the probability that the adversary wins in Game_3 is at most

$$\left(\frac{2L}{(L-1)N^2}\right)^{B-1}\cdot\frac{2}{L}=\frac{2^BL^{B-2}}{(L-1)^{B-1}N^{2B-2}}$$

For $L\geq 5$, $B\geq 2$, we have that $(L-1)^{B-1}\geq 2^B$ and so the probability that the adversary wins in Game_3 is at most

$$\frac{L^{B-2}}{N^{2B-2}}\leq\frac{1}{N^B},$$

as required.

Case 3 – $S=3$: Observing that $\binom{L}{3}=\frac{L(L-1)(L-2)}{6}$ and using the fact that $\binom{X}{t}^{-1}<\frac{L}{N}$ as above, in this case we have

$$\begin{aligned}&\Pr[\text{Game}_1(\mathcal{A},X,L,B,C)=1]\\ &< \left(\frac{6}{L(L-1)(L-2)}\right)^{B-1}\left(\frac{L}{N}\right)^{3(B-1)}\\ &= \left(\frac{6L^2}{(L-1)(L-2)N}\cdot\frac{1}{N^2}\right)^{B-1}\\ &= \left(\frac{6L}{(L-1)(L-2)(X-C)}\cdot\frac{1}{N^2}\right)^{B-1}\end{aligned}$$

By assumption $X>L+C$ and so $\frac{L}{X-C}\leq 1$, and $L\geq 5$ and so $\frac{6}{(L-1)(L-2)}\leq 1$. So the above is less than $\frac{1}{N^{2(B-1)}}$, which is at most $\frac{1}{N^B}$ for $B\geq 2$.

Case 4 – $S\geq 4$: Using the bound on Game_1 and again utilizing the fact that $\binom{X}{t}^{-1}<\frac{L}{N}$, first note that

$$\binom{L}{S}^{-1}\binom{X}{t}^{-S}\leq\left(\frac{L}{N}\right)^S\leq\frac{L^4}{N^4}=\frac{L^4}{N^2\cdot(X-C)^2L^2}$$

where the last equality is by definition that $N=(X-C)L$.

By the assumption that $X>L+C$ we have that $\frac{1}{X-C}<\frac{1}{L}$. Thus,

$$\binom{L}{S}^{-1}\binom{X}{t}^{-S}\leq\frac{L^4}{N^2L^4}=\frac{1}{N^2}.$$

Therefore

$$\begin{aligned}&\Pr[\text{Game}_1(\mathcal{A},X,L,B,C)=1]\\ &= \binom{L}{S}^{-(B-1)}\binom{X}{t}^{-S(B-1)}\leq\left(\frac{1}{N^2}\right)^{B-1}\leq\frac{1}{N^B}\end{aligned}$$

where the last inequality holds for $B\geq 2$. which suffices since $\mathcal{A}$ must win in Game_1 in order to win Game_3. ∎

Tightness of the theorem's bound. Note that in the above proof, in the case of $S=1$, $t=1$, the only inequality is $\frac{1}{X}<\frac{L}{N}$, i.e. $\frac{1}{X}<\frac{L}{(X-C)L}=\frac{1}{X-C}$. Since C is small, the bound for the case of $S=1$, $t=1$ is tight, and hence the bound

$$\Pr[\text{Game}_3(\mathcal{A},X,L,B,C)=1]\leq\frac{1}{N^B}$$

is tight.

An attack for $C=2$. We proved Theorem 3.10 for the case that $C\geq 3$. We conclude this section by showing that when $C=2$ the theorem does *not* hold and the adversary can win with probability $\frac{2}{N^2}$ (for $B=2$). The adversary works by corrupting no balls in the second array generated by Game_1 and by corrupting an entire subarray in the first array generated by Game_1. Specifically, in that execution of Game_1, it generates $X-2$ bad balls in some subarray in both arrays that it prepares. Since $C=2$, the probability that the adversary wins in Game_1 equals approximately $\frac{2L}{N^2}$. Then, in the later steps of Game_3 the adversary make the first ball b_1 bad and all the other balls good. Thus, the adversary wins if and only if b_1 is thrown into a bad bucket, which happens with probability $\frac{1}{L}$ (as there are $\frac{N}{L}$ bad buckets). Overall, the winning probability of the adversary is $\frac{2}{N^2}$.

A continuous version of Game_3. Next, we prove the bound for a continuous game which is equivalent to Protocol 3.9. This game runs indefinitely until $\mathcal{A}$ chooses a bad ball b_i in Step 3c. Note that the game terminates immediately after b_i is thrown in Step 3d: if b_i is thrown into a fully bad bucket, the output is 1, and if b_i is thrown into a fully good bucket, that bucket is no longer fully good and the output is 0. To ensure the game terminates, it is required that $\mathcal{A}$ choose a bad ball at some point.

$\underline{\text{Game}_3^{\text{C}}(\mathcal{A},X,L,B,C)}$:

1) Run steps 1-3 of $\text{Game}_1(\mathcal{A},X,L,B,C)$ once to receive a list of buckets $B_1,\ldots,B_N$.
2) If all buckets are either **fully good** or **fully bad** proceed to the next step. Otherwise, output 0.
3) Until an output is reached:
   a) Run steps 1-3 of $\text{Game}_1(\mathcal{A},X,L,B,C)$ once to receive a list of buckets $B'_1,\ldots,B'_N$.

b) If all buckets are either **fully good** or **fully bad** proceed to the next step. Otherwise, output 0.
c) The adversary $\mathcal{A}$ prepares N new balls denoted by $b_1, \ldots, b_N$, where each ball can be either **bad** or **good**.
d) For $i = 1$ to N:
   i) The ball b_i is thrown into a random bucket B_k ($k \in [N]$).
   ii) If the bucket B_k is **fully bad** output 1.
   iii) If the bucket B_k is not **fully good** or **fully bad** output 0.
   iv) Replace B_k with the bucket B'_i.

Theorem 3.11: Let $B \geq 2$ and assume $X > L+C$. Then for every adversary $\mathcal{A}$ and for every $L \geq 5$, $C \geq 3$ and $X-C \geq 6$ it holds that

$$\Pr[\text{Game}_3^{\text{C}}(\mathcal{A}, X, L, B, C) = 1] \leq \frac{1}{N^B}$$

where $N = (X - C)L$.

Proof: The only way for $\mathcal{A}$ to win is if a bad ball is thrown into a bad bucket in Step 3d. If the game ends prior to this step it will end with output 0. Therefore it is enough to show that when the first bad ball is thrown, $\mathcal{A}$ wins with probability at most $\frac{1}{N^B}$. There are three cases to consider.

- *Case 1:* A bad ball is thrown before any bad buckets are generated, that is, while all buckets $B_1, \ldots, B_N$ and $B'_1, \ldots, B'_N$ are good. In this case the output is always 0, hence the probability that $\mathcal{A}$ wins is 0.
- *Case 2:* When the first bad ball b_i is thrown, bad buckets have been generated during at least two executions of Game_1. By Theorem 3.4, $\mathcal{A}$ wins in two executions of Game_1 with probability $\frac{1}{N^{B-1}} \cdot \frac{1}{N^{B-1}} = \frac{1}{N^{2B-2}} \leq \frac{1}{N^B}$, where the last inequality holds since $B \geq 2$.
- *Case 3:* When the first bad ball b_i is thrown, bad buckets have been generated in exactly one execution of Game_1.
  - If the bad buckets are generated in Step 1 or the first iteration of Step 3a, the resulting game is equivalent to the case of Game_3 in which bad buckets are generated in exactly one list, and the bound follows from Theorem 3.10.
  - If, instead, the bad buckets are generated in a later iteration of Step 3a, note that $\mathcal{A}$ cannot win while all buckets are good, and the best strategy for $\mathcal{A}$ is to prepare no bad balls (since throwing a bad ball when all buckets are good always results in output 0). Thus any iteration of Step 3 in which no b_i is chosen will not increase the probability that $\mathcal{A}$ wins. Therefore, to obtain the bound, it is enough to consider only two iterations of Step 3: the iteration in which the bad ball b_i is produced, and the iteration preceding it. The resulting game is equivalent to the case of Game_3 in which bad buckets are generated in only the first list, and the bound follows from Theorem 3.10.

■

D. Hash Function Optimization

In [11], the method for validating a multiplication triple using another triple requires the parties to compare their views and verify that they are equal. In this basic comparison, each party sends 3 bits to another party. Since B such comparisons are carried out for every AND gate, this would significantly increase the communication. Concretely, with our parameters of $N = 2^{20}$ and $B = 2$ and our optimizations, this would increase the communication from 7 bits per AND gate to 13 bits per AND gate. In order to save this expense, [11] propose for each party to simply locally hash its view (using a collision-resistant hash function) and then to send the result of the hash only at the end of the protocol. Amortized over the entire computation, this would reduce this communication to almost zero. When profiling Protocol 3.9 with all of our optimizations, we were astounded to find that these hashes took up almost a *third* of the time in the triples-generation phase, and about 20% of the time in the circuit computation phase. Since the rate of computation is so fast, the SHA256 computations actually became a bottleneck; see Figure 3.

We solved this problem by observing that the view comparison procedure in [11] requires for each pair of parties to compare their view. The security is derived from the fact that if the adversary cheats then the views of the two *honest* parties are different. As such, instead of using a collision-resistant hash function, we can have each party compute a MAC of their view. In more detail, each pair of parties jointly choose a secret key for a MAC. Then, as the computation proceeds, each party computes a MAC on its view twice, once with each key for each other party. Then, at the end, each party sends the appropriate MAC to each other party. Observe that the honest parties compute a MAC using a secret key not known to the corrupted party. Thus, the adversary cannot cause the MACs of the two honest parties to have the same tag if their views are different (or this could be used to break the MAC). Note that with this method, each party computes the MAC on its view *twice*, in contrast to when using SHA256 where a single computation is sufficient. Nevertheless, we implemented this using GMAC (optimized using the PCLMULQDQ instruction) and the time spent on this computation was reduced to below 10%. As we show in Section IV, this method increases the throughput of the fastest protocol version by approximately 20%.

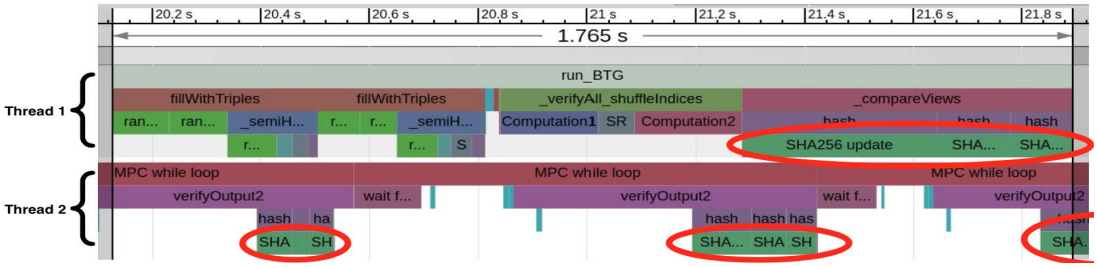

Fig. 3. Microbenchmarking of Protocol 3.9, using the CxxProf C++ profiler

IV. Implementation and Experimentation

We implemented the baseline protocol of [11] and the different protocol improvements and optimizations that we present in this paper. (We did not implement the variant in Section III-B since it has the same efficiency as the variant in Section III-C, and the latter is preferable for practical usage.) All of our implementations use parameters guaranteeing a cheating probability of at most 2^{-40}, as mandated by the appropriate theorem proven above. We begin by describing some key elements of our implementation, and then we present the experimental results.

A. Implementation Aspects

Parallelization and vectorization. As with [1], our protocol is particularly suited to vectorization. We therefore work in units of 256 bits, meaning that instead of using a single bit as the unit of operation, we perform operations on units of 256 bits simultaneously. For example, we are able to perform XOR operations on 256 bits at a time by writing a "for loop" of eight 32 bit integers. This loop is then automatically optimized by the Intel ICC compiler to use AVX2 256bit instructions (this is called *auto-vectorization*). We verified the optimization using the compiler vec-report flag and used `#pragma ivdep` in order to aid the compiler in understanding dependencies in the code. We remark that all of our combinatorial analyses considered "good" and "bad" balls and buckets. All of this analysis remains exactly the same when considering vectors of 256-triples as a single ball. This is because if any of the triples in a vector is bad, then this is detected and this is considered a "bad ball".

Memory management. We use a common data structure to manage large amounts of triplets in memory efficiently. This structure holds $2^{20} \times 256$ triplets. For triplets $([a], [b], [c])$ (or $([x], [y], [z])$ respectively) we store an array of $2^{20} \times 256$ bits for $[a]$, $2^{20} \times 256$ bits for $[b]$, and $2^{20} \times 256$ bits for $[c]$. This method is known as a *Struct of Arrays* (SoA) as opposed to an Array of Structs (AoS) and is commonly used in SIMD implementations. It provides for very efficient intrinsic (vectorized) operations, as well as fast communication since we send subarrays of these bit arrays over the communication channel in large chunks with zero memory copying. This reduces CPU cycles in the TCP/IP stack and is open for further optimization using RDMA techniques.

Index shuffling. When carrying out the shuffling, we shuffle indices of an indirection array instead of shuffling the actual triples (which are three 256-bit values and so 96 bytes). Later access to the 256-bit units is carried out by first resolving the location of the unit in $O(1)$ access to the indirection array. This show substantial improvement as this avoids expensive memory copies. Note that since the triples themselves are not shuffled, when reading the shuffled array during verification the memory access is not serial and we do not utilize memory prefetch and L3 cache. Nevertheless, our experiments show that this is far better overall than copying the three 256-bit memory chunks (96 bytes) when we shuffle data. In Figure 4, you can see that the entire cost of shuffling *and verifying* the triples (`_verifyAll_shuffleIndices`) is reduced to less than 30% of the time, in contrast to the original protocol in which it was approximately 55% (see Figure 1).

Cache-Aware code design. A typical Intel Architecture server includes a per-core L1 cache (32KB), a per-core L2 cache (typically 512KB to 2MB), and a CPU-wide L3 Cache (typically 25-55MB on a 20-36 core server). L1 cache access is extremely fast at $\sim$0.5ns, L2 access is $\sim$7ns and DDR memory reference is $\sim$100ns. All caches support write back (so updates to cached data is also extremely fast).

We designed our implementation to utilize L1 cache extensively when carrying out the Fisher-Yates shuffling on subarrays. We use two levels of indirection for the index shuffling: the top level of 512 indices and the low level of 2048 indices (under each of the top level indices, yielding 512 subarrays of length 2048 each). As vectors are 1024 byte and 4096 bytes respectively (`uint16` values), they require 1/32 or 1/8 of the L1 cache space so L1 will be utilized with very high probability (and in worst case will spill into the L2 cache). This makes shuffling extremely fast. Note that attempting to force prefetch of the index vectors into cache (using `_mm_prefetch` instructions) did not improve our performance, as this is hard to tune in real scenarios.

Offline/online. We implemented two versions of the protocols. The first version focuses on achieving high throughput and carries out the entire computation in parallel. Our best performance is achieved with 12 workers; each worker has two threads: the first thread generates multiplication triples, and the second carries out the circuit computation. The architecture of this version can be seen in Figure 5.

The second version focus on achieving fast online performance in an offline/online setting where multiplication triples are prepared ahead of time and then consumed later by a system running only the circuit computation (and verification of that computation). As we have mentioned, the cache-efficient version with bucket-size $B = 3$ is expected to have lower throughput than the version with bucket-size $B = 2$ but lower latency. This is because with $B = 3$ there is no need to randomly choose the triple being used to validate the gate being computed. We therefore compared these; note that in both cases we used the GMAC optimization described in Section III-D so that we would be comparing "best" versions.

TABLE II
Implementation results; B denotes the bucket size; security level 2^{-40}

Protocol Variant	AND gates/sec	%CPU utilization	Gbps utilization	Latency (ms)
Baseline [11]; Section II ($B = 3$, SHA)	503,766,615	71.7%	4.55	680
Cache-efficient; Sec. III-A ($B = 3$, SHA)	765,448,459	64.84%	7.28	623
On-demand; Sec. III-C ($B = 2$, SHA)	988,216,830	65.8%	6.84	812
On-demand; Sec. III-D ($B = 2$, GMAC)	**1,152,751,967**	71.28%	7.89	726
Online-only: on-demand; Sec. III-D ($B = 2$, GMAC)	1,726,737,312	45.1%	5.11	456.4
Online-only: cache-efficient; Sec. III-A ($B = 3$, GMAC)	**2,132,197,567**	41.6%	6.93	**367.5**

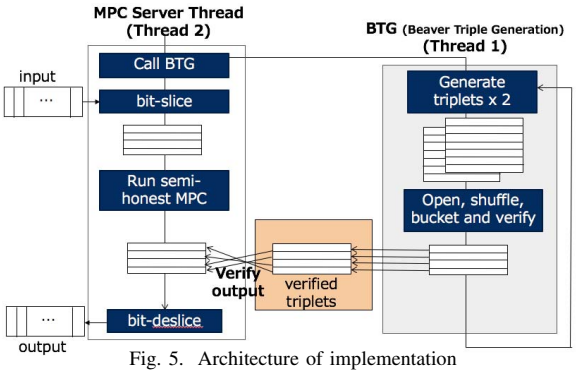

Fig. 5. Architecture of implementation

B. Experimental Results

We ran our implementations on a cluster of three mid-level servers connected by a 10Gbps LAN. Each server has two Intel Xeon E5-2650 v3 2.3GHz CPUs with a total of 20 cores. The results appear in Table II. Observe that each of the protocol improvements presented here provides a dramatic improvement:

- **Section III-A:** Replacing the naive Fisher-Yates shuffle on an array of size 2^{20} with our cache-efficient shuffle yields an increase of about 50% in throughput;
- **Section III-C:** Reducing the communication (in addition to the cache-efficient shuffle) by reducing the bucket-size from $B = 3$ to $B = 2$ and randomly choosing triples to verify the circuit multiplications yields a further increase of about 30%. (This is as expected since the reduction in communication is exactly 30%.)
- **Section III-D:** Replacing the use of SHA256 with the GMAC computations yielded an additional increase of over 15%.

Our best protocol version has a throughput of about ***2.3 times*** that of baseline version. This result unequivocally demonstrates that it is possible today to achieve ***secure computation with malicious adversaries at rates of well over 1-billion gates per second*** (using mid-level servers).

It is highly informative to also consider the results of the online-only experiments (where triples are prepared previously in an offline phase). As expected, the protocol version with bucket-size $B = 3$ is better in the online phase since no random choice of triples is needed. The throughput of the best version exceeds *2 billion AND gates per second*. Importantly, latency is also significantly reduced to 367.5ms; this can be important in some applications.

Microbenchmarks. Microbenchmarking of the faster protocol can be seen in Figure 4. In order to understand this, see Figure 5 for a description of the different elements in the implementation. The `run_BTG` thread generates multiplication (Beaver) triples. Each triple is generated by first generating two random sharings and then running a semi-honest multiplication. After two arrays of triples are prepared (since we use buckets of size $B = 2$), they are verified using the `_verifyAll_shuffleindices` procedure; this procedure carries out shuffling and verification. The second thread runs MPC computation to compute the circuit, followed by verifying all of the multiplications in the `verifyOutput2` procedure.

V. The Combinatorics of Cut-and-Choose

In the previous sections, we have seen that tight combinatorial analyses are crucial for practical performance. As pointed out in [11], the combinatorial analysis from [6] mandates a bucket-size of $B = 4$ for 2^{20} triples and security level $s = 2^{-40}$. In [11], a tighter combinatorial analysis enabled them to obtain the *same level of security* while reducing the bucket-size from $B = 4$ to $B = 3$. Utilizing a different method, we were further able to reduce the bucket size to $B = 2$. (Combinatorics also played an important role in achieving a cache-efficient shuffle and an on-demand version of the protocol.) With this understanding of the importance of combinatorics to cut-and-choose, in this section we ask some combinatorial questions that are of independent interest for cut-and-choose protocols.

A. The Potential of Different-Sized Buckets

We begin by studying whether the use of different-sized buckets can help to increase security. Since our Game$_1$ (from Section III-A) is specifically designed for the case where all buckets are of the same size, we go back to the more general game of [11] and [6] and redefine it so that buckets may have different sizes. Intuitively, since the adversary does not know in advance how many bad balls to choose so that there will be only fully bad buckets, using buckets of different sizes makes it more difficult for him to succeed in cheating. If this is indeed the case, then the winning probability of the adversary can be further decreased, and it may be possible to generate less triples to start with, further improving efficiency. In [11, Theorem 5.3] it was shown that the optimal strategy for the adversary is to make the number of bad balls equal to the size of a single bucket. In this section, we show that even when the buckets sizes are different, the best strategy for the adversary

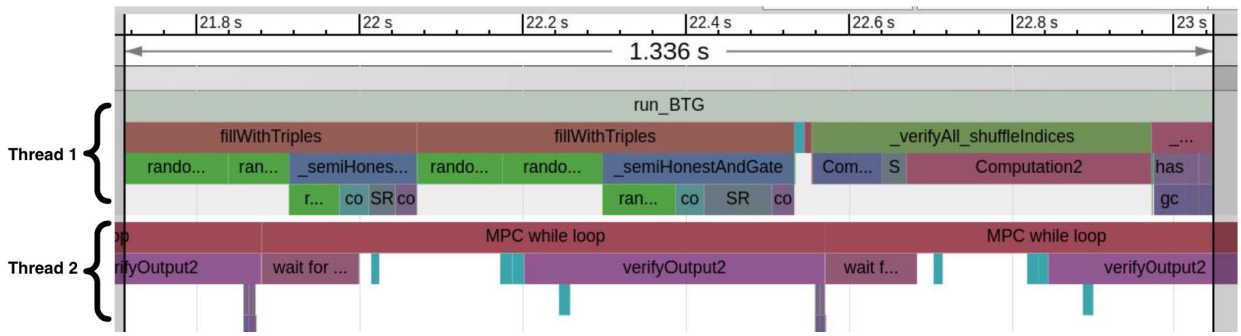

Fig. 4. Microbenchmarking of best protocol variant, using the CxxProf C++ profiler (run on a local host)

is to make the number of bad balls equal to the size of the smallest bucket. We then use this fact to show that given any set of bucket sizes, changing the sizes so that the bucket sizes of any two buckets differ by at most 1, does not improve the probability that the adversary wins. This makes sense since the adversary's best strategy is to make the number of bad balls equal the size of the smallest bucket, and its hope is that the bad balls will fall into such a bucket. By reducing the gap between buckets (by moving balls from larger buckets to smaller ones) we actually reduce the number of buckets of the smallest size, thereby reducing the probability that all bad balls will be in a bucket of minimal size.

We define a combinatorial game with buckets of different sizes as follows. Let $\vec{B} = \{B_1, \ldots, B_N\}$ denote the multiset of bucket sizes where B_i is the size of the ith bucket. As C balls are opened before dividing the balls into buckets, it follows that the overall number of balls generated is $M = \sum_{i=1}^{N} B_i + C$.

$\text{Game}_4(\mathcal{A}, N, \vec{B}, C)$**:**

1) The adversary $\mathcal{A}$ prepares M balls. Each ball can be either **bad** or **good**.
2) C random balls are chosen and opened. If one of the C balls is **bad** then output 0. Otherwise, the game proceeds to the next step.
3) The remaining $\sum_{i=1}^{N} B_i$ balls are randomly thrown into N buckets of sizes $\vec{B} = \{B_1, \ldots, B_N\}$.
4) The output of the game is 1 if and only if there exists a bucket B_i that is **fully bad**, and all other buckets are either **fully bad** or **fully good**.

For our analysis we need some more notation. Let B_{min} be the minimal bucket size. We use $[N]$ to denote the set $\{1, \ldots, N\}$. Let $S \subseteq [N]$ be a subset of bucket indices, and let $t_S = \sum_{\{i|i \in S\}} B_i$ be the total number of balls in the buckets indexed by S. Finally, let $n(t) = |\{S \subseteq [N] \mid t_S = t\}|$ be the number of different subsets of buckets such that the number of balls in all buckets in the subset equal exactly t.

We start by computing the winning probability of the adversary. First, we prove that a necessary and sufficient condition for $\mathcal{A}$ having any chance to win in the game, is that $n(t) > 0$.

Claim 5.1: Let $\mathcal{A}_t$ be an adversary who chooses t bad balls. Then, $\Pr[\text{Game}_4(\mathcal{A}_t, N, \vec{B}, C) = 1] > 0$ if and only if $n(t) > 0$.

Proof: If $n(t) = 0$ then there is no subset of buckets which contain exactly t balls. Thus, every permutation of the balls will result in the existence of a mixed bucket containing good and bad balls, and the output of the game will be 0 with probability 1. In contrast, if $n(t) > 0$ then there exists a subset $S \subseteq \vec{B}$ such that $t_S = t$ and therefore with non-zero probability, the bad balls will fall only in the buckets of S, and the game's output will be 1. ∎

Intuitively, for $\mathcal{A}$ to win, the bad balls must fill some subset of buckets (since otherwise there will be a bucket with good and bad balls). Since there are $n(t)$ such subsets, and there are $\binom{M}{t}$ ways to choose t balls out of M balls, it follows that the winning probability of the adversary is $\frac{n(t)}{\binom{M}{t}}$ as stated in the next lemma.

Lemma 5.2: For every adversary $\mathcal{A}_t$ who chooses t bad balls it holds that

$$\Pr[\text{Game}_4(\mathcal{A}_t, N, \vec{B}, C) = 1] = \frac{n(t)}{\binom{M}{t}}.$$

Proof: First, the probability that $\mathcal{A}$ does not lose when C balls are opened is $\frac{\binom{M-t}{C}}{\binom{M}{C}}$, since there are $\binom{M}{C}$ ways to choose C balls overall, and $\binom{M-t}{C}$ ways to choose C good balls. Then, there are $(M-C)!$ ways to permute the balls, from which only $n(t) \cdot t! \cdot (M-C-t)!$ will result in $\mathcal{A}$ winning the game. This holds since $n(t)$ equals the number of subsets of buckets which contain exactly t balls, there are $t!$ ways to permute the bad balls inside the buckets in such a subset and there are $(M-C-t)!$ ways to permute the remaining balls. Therefore, we obtain that

$$\begin{aligned}
&\Pr[\text{Game}_4(\mathcal{A}_t, N, \vec{B}, C) = 1] \\
&= \frac{\binom{M-t}{C}}{\binom{M}{C}} \cdot \frac{n(t) \cdot t!(M-C-t)!}{(M-C)!} \\
&= n(t) \cdot \frac{(M-t)!}{C!(M-t-C)!} \cdot \frac{C!(M-C)!}{M!} \cdot \frac{t!(M-C-t)!}{(M-C)!} \\
&= n(t) \cdot \frac{t!(M-t)!}{M!} = \frac{n(t)}{\binom{M}{t}}
\end{aligned}$$

∎

The next theorem proves that the best strategy for the adversary is to corrupt a single bucket of minimal size. The intuition behind this, is that in order for a subset of buckets to be filled with t bad balls, the smallest bucket in this subset must be filled with bad balls. Thus, it is better for the adversary to choose bad balls for this bucket only, instead for the entire subset.

Theorem 5.3: If $C \geq B_{min}$ then for every $S \subseteq [N]$, for every adversary $\mathcal{A}_{t_S}$ who chooses t_S bad balls and for every adversary $\mathcal{A}_{B_{min}}$ who chooses B_{min} bad balls, it holds that

$$\begin{aligned}
&\Pr[\text{Game}_4(\mathcal{A}_{t_S}, N, \vec{B}, C) = 1] \\
&\qquad \leq \Pr[\text{Game}_4(\mathcal{A}_{B_{min}}, N, \vec{B}, C) = 1].
\end{aligned}$$

We begin by proving a simple property of $n(t)$ that will be used later in the proof of Theorem 5.3.

Claim 5.4: For every M, C as defined in Game_4 and for every $0 < t < M - C$ it holds that $n(t) = n(M - C - t)$.

Proof: This follows directly from the definition. Specifically, for every subset of buckets S such that $t_S = t$, the complement subset $\overline{S} = \vec{B} \setminus S$ is such that $t_{\overline{S}} = M - C - t$. Thus, the number of subsets is the same. ∎

Next, we prove a bound on the winning probability of the adversary.

Lemma 5.5: For every $S \subseteq [N]$ such that $t_S \leq \frac{M}{2}$ and for every adversary $\mathcal{A}_{t_S}$ who chooses t_S bad balls, there exists $r \in \left[\frac{N}{2}\right]$ for which it holds that

$$\Pr[\text{Game}_4(\mathcal{A}_{t_S}, N, \vec{B}, C) = 1] \leq \frac{n(t_S)}{\binom{M}{r \cdot B_{min}}}$$

Proof: From Lemma 5.2, it follows that it is sufficient to show that there exists $r \in \left[\frac{N}{2}\right]$ for which it holds that

$$\frac{1}{\binom{M}{t_S}} \leq \frac{1}{\binom{M}{r \cdot B_{min}}}$$

which is equivalent to

$$\binom{M}{t_S} \geq \binom{M}{r \cdot B_{min}}. \tag{5}$$

We consider two cases (note that $|S|$ is the number of buckets in the subset S):

- *Case 1*: $|S| \leq \frac{N}{2}$. In this case, we simply set $r = |S|$ and then the lemma holds since $1 \leq r \leq \frac{N}{2}$ and since $r \cdot B_{min} = |S| \cdot B_{min} \leq t_S \leq \frac{M}{2}$ (which implies that Eq. (5) holds, since the expression $\binom{M}{x}$ increases as x increases when $x \in \{1, \ldots, \frac{M}{2}\}$).
- *Case 2*: $|S| > \frac{N}{2}$. In this case, we set $r = \frac{N}{2}$. Then, the lemma holds since $r \in \left[\frac{N}{2}\right]$ and since $\frac{N}{2} \cdot B_{min} < |S| \cdot B_{min} \leq t_S \leq \frac{M}{2}$ (which, as in the previous case, implies that Eq. (5) holds).

■

We are now ready to prove that the best strategy for the adversary is to choose exactly B_{min} bad balls; i.e., to corrupt a single bucket of minimal size.

Proof of Theorem 5.3: From Lemma 5.2 it follows that

$$\Pr[\text{Game}_4(\mathcal{A}_{B_{min}}, N, \vec{B}, C) = 1] = \frac{n(B_{min})}{\binom{M}{B_{min}}}$$

We consider first the case where $t_S \leq \frac{M}{2}$. From Lemma 5.5 it follows that in this case, there exists $r \in [\frac{N}{2}]$ such that

$$\Pr[\text{Game}_4(\mathcal{A}_{t_S}, N, \vec{B}, C) = 1] \leq \frac{n(t_s)}{\binom{M}{r \cdot B_{min}}}.$$

Thus, in order to prove the theorem, we need to show that

$$\frac{n(t_s)}{\binom{M}{r \cdot B_{min}}} \leq \frac{n(B_{min})}{\binom{M}{B_{min}}}.$$

Since $n(B_{min}) > 0$, it suffices to prove that

$$\frac{n(t_s)}{\binom{M}{r \cdot B_{min}}} \leq \frac{1}{\binom{M}{B_{min}}}$$

which is equivalent to

$$n(t_S) \cdot \frac{M!}{(M - B_{min})! B_{min}!} \leq \frac{M!}{(M - r \cdot B_{min})!(r \cdot B_{min})!}.$$

By multiplying both sides with

$$\frac{B_{min}!(M - r \cdot B_{min})!(r \cdot B_{min} - B_{min})!}{M!},$$

we can replace the above inequality with

$$\begin{aligned} n(t_S) \cdot \frac{(M - r \cdot B_{min})!(r \cdot B_{min} - B_{min})!}{(M - B_{min})!} \\ \leq \quad \frac{B_{min}!(r \cdot B_{min} - B_{min})!}{(r \cdot B_{min})!} \end{aligned}$$

which is equivalent to proving that

$$\begin{aligned} n(t_S) \cdot \frac{(r \cdot B_{min})!}{B_{min}!(r \cdot B_{min} - B_{min})!} \\ \leq \quad \frac{(M - B_{min})!}{(r \cdot B_{min} - B_{min})!(M - r \cdot B_{min})!}. \end{aligned}$$

Thus, we need to prove that

$$n(t_S)\binom{r \cdot B_{min}}{(r-1) \cdot B_{min}} \leq \binom{M - B_{min}}{(r-1) \cdot B_{min}}.$$

Since $C \geq B_{min}$, it holds that $M - B_{min} \geq M - C$ and so

$$\binom{M - B_{min}}{(r-1) \cdot B_{min}} \geq \binom{M - C}{(r-1) \cdot B_{min}}.$$

In addition, note that $n(t_S) \leq \binom{M-C}{t_S}$ (this holds since by definition $n(t_S)$ equals the number of ways to choose t_S balls out of $M - C$ balls under *a specific restriction*), and note that $r \cdot B_{min} \leq t_S$ (see Lemma 5.5). Thus it follows that

$$n(t_S)\binom{r \cdot B_{min}}{(r-1) \cdot B_{min}} \leq \binom{M - C}{t_S}\binom{t_S}{(r-1) \cdot B_{min}}.$$

Combining all above, it suffices to prove that

$$\binom{M - C}{t_S}\binom{t_s}{(r-1) \cdot B_{min}} \leq \binom{M - C}{(r-1) \cdot B_{min}}. \tag{6}$$

To see that Eq. (6) holds, consider the following two combinatorial processes: **(1)** choose t_S balls out of $M - C$ and then choose $(r-1) \cdot B_{min}$ balls from the t_S that were chosen before; **(2)** choose $(r-1) \cdot B_{min}$ balls out of $M - C$ balls. It is easy to see that the number of ways to choose the t_S balls is higher in the second process as the selection of balls is less restricted. Since the first process corresponds to the left side of Eq. (6) whereas the second process corresponds to the right side, we conclude that the theorem holds in this case.

Next, we proceed to the second case where $t_S \geq \frac{M}{2}$.

Since $t_S \geq \frac{M}{2}$ and $t_S < C + t_S$ it holds that $\binom{M}{t_S} > \binom{M}{t_S + C}$ (this holds since $\binom{M}{x}$ decreases as x increases when $x \in \{\frac{M}{2}, \ldots, M\}$). Thus, it holds that

$$\binom{M}{t_S} > \binom{M}{t_S + C} = \binom{M}{M - C - t_S}. \tag{7}$$

Therefore, it holds that

$$\begin{aligned} &\Pr[\text{Game}_4(\mathcal{A}_{t_S}, N, \vec{B}, C) = 1] \\ &= \frac{n(t_S)}{\binom{M}{t_S}} < \frac{n(M - C - t_S)}{\binom{M}{M - C - t_S}} \\ &= \Pr[\text{Game}_4(\mathcal{A}_{M-C-t_S}, N, \vec{B}, C) = 1] \end{aligned}$$

where the first and last equalities holds from Lemma 5.2, and the inequality follows from Eq. (7) and Claim 5.4, which states that $n(t_S) = n(M - C - t_S)$.

Now, since $M - C - t_S < \frac{M}{2}$, it follows from the first case that

$$\begin{aligned} &\Pr[\text{Game}_4(\mathcal{A}_{M-C-t_S}, N, \vec{B}, C) = 1] \\ &\leq \Pr[\text{Game}_4(\mathcal{A}_{B_{min}}, N, \vec{B}, C) = 1] \end{aligned}$$

and therefore

$$\begin{aligned}&\Pr[\text{Game}_4(\mathcal{A}_{t_S}, N, \vec{B}, C) = 1] \\ &\quad \leq \Pr[\text{Game}_4(\mathcal{A}_{B_{min}}, N, \vec{B}, C) = 1]\end{aligned}$$

as required. This concludes the proof of the theorem. ∎

Next, we show that if $\vec{B}$ that was chosen for the game contains two buckets i and j such that $B_i - B_j > 1$, then moving one ball from the bigger bucket B_i to the smaller bucket B_j, will result in a game that is more difficult for the adversary to win. This proves that having buckets of significantly different sizes does not improve security, as one can keep moving balls between buckets until all buckets are of size B and $B+1$ for some B. As explained earlier, the intuition behind this is that reducing the gap between large and small buckets in this way can only result in having fewer buckets of smallest size, and therefore the probability that the bad balls will be thrown into a bucket of smallest size can only be reduced.

Theorem 5.6: Let $\vec{B}$ be a multiset of N bucket sizes that was chosen for the game and assume that there exist $i, j \in [N]$ such that $B_i - B_j > 1$. Let $\vec{B}'$ be a multiset of N bucket sizes obtained by setting

$$B'_k = \begin{cases} B_i - 1 & \text{if } k = i \\ B_j + 1 & \text{if } k = j \\ B_k & \text{otherwise} \end{cases}$$

If $C \geq B_{min}$, then for every adversary $\mathcal{A}'$ in the game where $\vec{B}'$ is used, there exists an adversary $\mathcal{A}$ in the game where $\vec{B}$ is used such that

$$\Pr[\text{Game}_4(\mathcal{A}', N, \vec{B}', C) = 1] \leq \Pr[\text{Game}_4(\mathcal{A}, N, \vec{B}, C) = 1]$$

Proof: Denote by B_{min} the size of the smallest bucket in $\vec{B}$ and by B'_{min} the size of the smallest bucket in $\vec{B}'$. From Theorem 5.3, it follows that for every adversary $\mathcal{A}'$ as in the lemma it holds that

$$\begin{aligned}&\Pr[\text{Game}_4(\mathcal{A}', N, \vec{B}', C) = 1] \\ &\quad \leq \Pr[\text{Game}_4(\mathcal{A}'_{B'_{min}}, N, \vec{B}', C) = 1]\end{aligned}$$

where $\mathcal{A}'_{B'_{min}}$ is an adversary who chooses B'_{min} bad balls in the game where $\vec{B}'$ is used. It suffices to show that

$$\begin{aligned}&\Pr[\text{Game}_4(\mathcal{A}'_{B'_{min}}, N, \vec{B}', C) = 1] \\ &\quad \leq \Pr[\text{Game}_4(\mathcal{A}_{B_{min}}, N, \vec{B}, C) = 1] \qquad (8)\end{aligned}$$

since we can then take $\mathcal{A} = \mathcal{A}_{B_{min}}$ in the game with $\vec{B}'$ in the theorem statement.

From Lemma 5.2 we obtain that Eq. (8) can be replaced by

$$\frac{n(B'_{min})}{\binom{M}{B'_{min}}} \leq \frac{n(B_{min})}{\binom{M}{B_{min}}}. \qquad (9)$$

To prove Eq. (9), we consider two cases:

- *Case 1*: $B'_{min} = B_{min}$. In this case, it is sufficient to prove that $n(B'_{min}) \leq n(B_{min})$, and this holds since in the process of changing $\vec{B}$ to $\vec{B}'$, the number of buckets with smallest size can only decrease (note that since only one bucket can be filled when $t = B_{min}$, then $n(B_{min})$ equals the number of buckets of minimal size).
- *Case 2*: $B'_{min} > B_{min}$. This case can occur only if there was exactly one bucket of size B_{min} in $\vec{B}$, and it gained one ball in the process of changing $\vec{B}$ to $\vec{B}'$. Thus, it follows that $B'_{min} = B_{min} + 1$ and $n(B_{min}) = 1$. Thus, we can replace Eq. (9) with

$$n(B'_{min}) \leq \frac{\binom{M}{B'_{min}}}{\binom{M}{B_{min}}}.$$

Observe that

$$\begin{aligned}\frac{\binom{M}{B'_{min}}}{\binom{M}{B_{min}}} &= \frac{(M - B_{min})!B_{min}!}{(M - B'_{min})!B'_{min}!} \\ &= \frac{(M - B_{min})!B_{min}!}{(M - (B_{min}+1))!(B_{min}+1)!} \\ &= \frac{M - B_{min}}{B_{min}+1} \geq \frac{M - C}{B_{min}+1} = \frac{\sum_{i=1}^{N} B_i}{B_{min}+1}\end{aligned}$$

where the inequality holds since $C \geq B_{min}$. Also, note that $n(B_{min}) \leq N$ as there are N buckets in the game. Thus, in order to complete the proof, we need to show that

$$N \leq \frac{\sum_{i=1}^{N} B_i}{B_{min}+1}.$$

This holds since, as explained before, in this case there was exactly one bucket of size B_{min}, one bucket of size at least $B_{min}+2$ (from which one ball was moved to the first bucket) and all other buckets are of size at least $B_{min}+1$ (otherwise, B'_{min} would have stayed equal to B_{min}). Thus, it follows that

$$\begin{aligned}\frac{\sum_{i=1}^{N} B_i}{B_{min}+1} &\geq \frac{B_{min} + (B_{min}+2) + (N-2)(B_{min}+1)}{B_{min}+1} \\ &= \frac{N \cdot (B_{min}+1)}{B_{min}+1} = N\end{aligned}$$

as required.

Note that these are the only possible cases, since the case that $B'_{min} < B_{min}$ is not possible, because a bucket of minimal size cannot lose balls when modifying $\vec{B}$ to obtain $\vec{B}'$. This concludes the proof of the theorem. ∎

We conclude that taking different-sized buckets does not improve security (except possibly for the case when exactly two sizes B and $B+1$ are used). We will use this conclusion in the next section.

B. Moderately Lowering the Cheating Probability

The discrete cut-and-choose problem. Typically, when setting the parameters of a protocol that has statistical error (like in cut and choose), there is a targeted "allowed" cheating probability which determines a range of values that guarantee the security bound. The parameters are then chosen to achieve the best efficiency possibly within the given range. For example, in a cut-and-choose setting modeled with balls and buckets, the size of the buckets B may be incremented until the security bound is met. However, this strategy can actually be very wasteful. In order to understand why, assume that the required security bound is 2^{-40} and assume that for the required number of buckets, the bound obtained when setting $B = 3$ is 2^{-39}. Since this is above the allowed bound, it is necessary to increase the bucket size to $B = 4$. This has the effect of increasing the protocol complexity significantly while reducing the security bound to way below what is required. To be concrete, we have proven that the error bound

for the protocol version in Section III-A in $1/N^{B-1}$ (see Theorem 3.4). If we require a bound of 2^{-40} and wish to carry out $N = 2^{19} \approx 500,000$ executions, then with $B = 3$ we achieve a cheating probability of only 2^{-38}. By increasing the bucket size to $B = 4$ we obtain a bound of 2^{-57} which is overkill with respect to the desired bound. It would therefore be desirable to have a method that enables us to trade-off the protocol complexity and cheating probability in a more fine-grained manner.

A solution. In this section, we propose a partial solution to this problem; our solution is only partial since it is not as fine-grained as we would like. Nevertheless, we view this as a first step to achieving better solutions to the problem. The solution that we propose in this section is to increment the size of only *some* of the buckets by 1 (instead of all of them), resulting in a game where there are buckets of two sizes, B and $B+1$. We use the analysis of the previous section to show that this gradually reduces to the error probability, as desired.

Formally, let $\vec{B}^k = \{B_1^k, \ldots, B_N^k\}$ be a multiset of bucket sizes such that $B_i^k = B$ for $i \leq k$ and $B_i^k = B+1$ for $i > k$. In the next lemma, we show that the probability that the adversary wins in the combinatorial game when choosing the bucket sizes in this way is a multiplicative factor of $p = \frac{k}{N}$ lower than when all buckets are of size B. Thus, in order to reduce the probability by $1/2$, it suffices to take $k = N/2$ and increase half the buckets to size $B+1$ instead of all of them. In the concrete example above, with $N = 2^{19}$ it is possible to reduce the bound to 2^{-40} by increasing half of the buckets to size $B = 4$ instead of all of them, achieving a saving of 2^{18} balls. This therefore achieves the desired goal. We now prove the lemma.

Lemma 5.7: Let $k, N \in \mathbb{N}$ such that $k < N$ and let $p = \frac{k}{N}$. For every bucket-size B, let $\vec{B}^k$ be the multiset of bucket sizes defined as above. Then, for every adversary $\mathcal{A}^k$ in Game_4 where $\vec{B}^k$ is used, there exists an adversary $\mathcal{A}$ in Game_4 where all buckets are of size B such that

$$\Pr[\text{Game}_4(\mathcal{A}^k, N, \vec{B}^k, C) = 1] \leq p \cdot \Pr[\text{Game}_4(\mathcal{A}, N, B, C) = 1].$$

Proof: In the version of Game_4 with bucket sizes $\vec{B}^k$, the minimal bucket size is B. Thus using Theorem 5.3, an adversary who chooses B bad balls will maximize its winning probability in both games. Thus, it is sufficient to prove that

$$\begin{aligned} &\Pr[\text{Game}_4(\mathcal{A}_B^k, N, \vec{B}^k, C) = 1] \\ &\quad \leq \quad p \cdot \Pr[\text{Game}_4(\mathcal{A}_B, N, B, C) = 1] \end{aligned} \tag{10}$$

where $\mathcal{A}_B$ and $\mathcal{A}_B^k$ are adversaries who choose B bad balls in their games. This is sufficient since if Eq. (10) holds then for every $\mathcal{A}^k$, we can take the adversary $\mathcal{A}_B$ as the adversary for which the lemma holds.

Since there are exactly k buckets of size B, we have that $n(B) = k$ in this game. Furthermore, the number of balls overall is exactly $Bk + (B+1)(N-k) + C$. Thus, by Lemma 5.2, it holds that

$$\begin{aligned} \Pr[\text{Game}_4(\mathcal{A}_B^k, N, \vec{B}^k, C) = 1] &= \frac{k}{\binom{Bk+(B+1)(N-k)+C}{B}} \\ &= \frac{p \cdot N}{\binom{BN+(N-k)+C}{B}}. \end{aligned}$$

Similarly, from Lemma 5.2, it follows that

$$\Pr[\text{Game}_4(\mathcal{A}_B, N, B, C) = 1] = \frac{N}{\binom{BN+C}{B}}$$

since there are N buckets of size B in this game. Thus, Eq. (10) follows if

$$\frac{p \cdot N}{\binom{BN+(N-k)+C}{B}} \leq \frac{p \cdot N}{\binom{BN+C}{B}}$$

and this holds since $\binom{BN+C}{B} \leq \binom{BN+(N-k)+C}{B}$. ∎

Improving the bound. Observe that the adversary's winning probability decreases multiplicatively by k/N when $N-k$ balls are added. Thus, in order to reduce the probability by $1/2$ we must add $N - N/2 = N/2$ balls, and in order to reduce the probability by $1/4$ we must add $3N/4$ balls. In general, in order to reduce the probability by $2^{-\zeta}$ we must add $N - N/2^{\zeta}$ balls. An important question that is open is whether or not it is possible to reduce the probability while adding fewer balls.

ACKNOWLEDGEMENTS

The authors at Bar-Ilan University were supported in part 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.

REFERENCES

[1] T. Araki, J. Furukawa, Y. Lindell, A. Nof and K. Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In the 23*rd ACM CCS*, pages 805–817, 2016.

[2] D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO 1991*, Springer (LNCS 576), pages 420–432, 1992.

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

[4] 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.

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

[6] S.S. Burra, E. Larraia, J.B. Nielsen, P.S. Nordholt, C. Orlandi, E. Orsini, P. Scholl, and N.P. Smart. High Performance Multi-Party Computation for Binary Circuits Based on Oblivious Transfer. *ePrint Cryptology Archive*, 2015/472.

[7] D. Chaum, C. Crépeau and I. Damgård. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.

[8] I. Damgård, M. Geisler, M. Krøigaard and J.B.Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *Public Key Cryptography 2009*, Springer (LNCS 5443), pages 160–179, 2009.

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

[10] R.A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research* (3rd ed.). Oliver & Boyd. pages 26-27, 1938.

[11] J. Furukawa, Y. Lindell, A. Nof and O. Weinstein. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT 2017*.

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

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

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

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

[16] M. Keller, E. Orsini and P. Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *ACM CCS*, pages 830–842, 2016.

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

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

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

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

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

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

[23] P. Rindal and M. Rosulek. Faster Malicious 2-Party Secure Computation with Online/Offline Dual Execution. In *USENIX Security Symposium*, pages 297–314, 2016.

[24] 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.

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

[26] 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.

[27] Intel Haswell cache performance. `http://www.7-cpu.com/cpu/Haswell.html`

APPENDIX A
THE PROTOCOL OF [11] IN DETAIL

PROTOCOL A.1 (Generating Multiplication triples):

- **Input:** The number N of triples to be generated.
- **Auxiliary input:** Parameters B and C.
- **The Protocol:**
  1) *Generate random sharings:* The parties generate $2M$ sharings of random values, where $M = 2(NB + C(B-1))$; denote the shares that they receive by $[([a_i],[b_i])]_{i=1}^{M}$.
  2) *Generate multiplication triples:* For $i = 1,\ldots,M$, the parties run the semi-honest multiplication protocol of [1] to compute $[c_i] = [a_i] \cdot [b_i]$. Denote $\vec{D} = [([a_i],[b_i],[c_i])]_{i=1}^{M/2}$; observe that $[c_i]$ is the result of the protocol and is not necessarily "correct".
  3) *Cut and bucket:* In this stage, the parties perform a first verification that the triples were generated correctly, by opening some of the triples.
     a) Each party splits $\vec{D}$ into vectors $\vec{D}_1,\ldots,\vec{D}_B$ such that $\vec{D}_1$ contains N triples and each $\vec{D}_i$ for $i = 2,\ldots,B$ contains $N + C$ triples.
     b) For $i = 2$ to B: The parties jointly and securely generate a random permutation π_i over $\{1,\ldots,N+C\}$ and then each locally shuffle $\vec{D}_i$ according to π_i.
     c) For $i = 2$ to B: The parties run a protocol for checking that a triple is valid (with opening) for the first C triples in $\vec{D}_i$, and removes them from $\vec{D}_i$. If a party did not output accept in every execution, it sends $\perp$ to the other parties and outputs $\perp$.
     d) The remaining triples are divided into N sets of triples $\vec{E}_1,\ldots,\vec{E}_N$, each of size B, such that the bucket $\vec{E}_i$ contains the i'th triple in $\vec{D}_1,\ldots,\vec{D}_B$.
  4) *Check buckets:* The parties initialize a vector $\vec{d}$ of length N. Then, for $i = 1,\ldots,N$:
     a) Denote the triples in $\vec{E}_i$ by $([a_1],[b_1],[c_1]),\ldots,([a_B],[b_B],[c_B])$.
     b) For $j = 2,\ldots,B$, the parties run a protocol to check that $([a_1],[b_1],[c_1])$ is valid (i.e., $c_1 = a_1 b_1$), using $([a_j],[b_j],[c_j])$.
     c) If a party did not output accept in every execution, it sends $\perp$ to the other parties and outputs $\perp$.
     d) The parties set $\vec{d}_i = ([a_1],[b_1],[c_1])$; i.e., they store these shares in the ith entry of $\vec{d}$.
- **Output:** The parties output $\vec{d}$.

PROTOCOL A.2 (Securely Computing a Functionality f):

- **Inputs:** Each party P_i where $i \in \{1,2,3\}$ holds an input $x_i \in \{0,1\}^{\ell}$.
- **Auxiliary Input:** The parties hold a description of a Boolean circuit $\mathcal{C}$ that computes f on inputs of length ℓ. Let N be the number of gates in $\mathcal{C}$.
- **The protocol – offline phase:**
  1) The parties call Protocol A.1 with input N and obtain a vector $\vec{d}$ of sharings.
- **The protocol – online phase:**
  1) *Sharing the inputs:* For each input wire, the party whose input is associated with that wire securely shares its input.
  2) *Circuit emulation:* Let $G_1,\ldots,G_N$ be a predetermined topological ordering of the gates of the circuit. For $k = 1,\ldots,N$ the parties work as follows:
     – If G_k is a XOR gate: Given shares $[x]$ and $[y]$ on the input wires, the parties compute $[x] \oplus [y]$ and define the result as their share on the output wire.
     – If G_k is a NOT gate: Given shares $[x]$ on the input wire, the parties compute $\overline{[x]}$ and define the result as their share on the output wire.
     – If G_k is an AND gate: Given shares $[x]$ and $[y]$ on the input wires, the parties run the semi-honest multiplication protocol of [1].
  3) *Verification stage: Before* the shared values on the output wires are reconstructed, the parties verify that all the multiplications were carried out correctly, as follows. For $k = 1,\ldots,N$:
     a) Denote by $([x],[y])$ the shares of the input wires to the kth AND gate, and denote by $[z]$ the shares of the output wire of the kth AND gate.
     b) The parties run the protocol to check that the triple $([x],[y],[z])$ is valid (i.e., $z = xy$) using the triple $([a_k],[b_k],[c_k])$.
     c) If a party did not output accept in every execution, it sends $\perp$ to the other parties and outputs $\perp$.
  4) If any party received $\perp$ in any call to any functionality above, then it outputs $\perp$ and halts.
  5) *Output reconstruction:* For each output wire of the circuit, the parties securely reconstruct the shared secret to the party whose output is on the wire.
  6) If any party receives $\perp$ in any such reconstruction, then it sends $\perp$ to the other parties, outputs $\perp$ and halts.
- **Output:** If a party has not output $\perp$, then it outputs the values it received on its output wires.

In this appendix, we describe the protocol of [11] in detail. We omit the description of the secret-sharing scheme, the semi-honest multiplication protocol and other details that are not needed for understanding our techniques. The description below also refers to subroutines for generating shares of random values, for jointly choosing a random permutation (for the array shuffle), and for sharing and reconstructing secrets. In addition, they use subprotocols for checking the validity of a multiplication triple (while opening and "wasting it") and for checking the validity of triple using another (while preserving the secrecy of the first and "wasting" the second). The protocol specifications below are as in [11], with the exception of the cut-and-bucket step which is modified as described in Section II-A.