High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority

Toshinori Araki
NEC Corporation, Japan
t-araki@ek.jp.nec.com

Jun Furukawa
NEC Corporation, Japan
j-furukawa@ay.jp.nec.com

Yehuda Lindell*
Bar-Ilan University, Israel
lindell@biu.ac.il

Ariel Nof*
Bar-Ilan University, Israel
nofdinar@gmail.com

Kazuma Ohara
NEC Corporation, Japan
k-ohara@ax.jp.nec.com

ABSTRACT

In this paper, we describe a new information-theoretic protocol (and a computationally-secure variant) for secure *three*-party computation with an honest majority. The protocol has very minimal computation and communication; for Boolean circuits, each party sends only a single bit for every AND gate (and nothing is sent for XOR gates). Our protocol is (simulation-based) secure in the presence of semi-honest adversaries, and achieves privacy in the client/server model in the presence of malicious adversaries.

On a cluster of three 20-core servers with a 10Gbps connection, the implementation of our protocol carries out over *1.3 million* AES computations per second, which involves processing over *7 billion gates per second*. In addition, we developed a Kerberos extension that replaces the ticket-granting-ticket encryption on the Key Distribution Center (KDC) in MIT-Kerberos with our protocol, using keys/ passwords that are shared between the servers. This enables the use of Kerberos while protecting passwords. Our implementation is able to support a login storm of over 35,000 logins per second, which suffices even for very large organizations. Our work demonstrates that high-throughput secure computation is possible on standard hardware.

1. INTRODUCTION

1.1 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 to be 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). In the *information-theoretic* model, security is obtained unconditionally and even in the presence of computationally unbounded adversaries. In contrast, in the *computational* model, security is obtained in the presence of polynomial-time adversaries and relies on cryptographic hardness assumptions.

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 [2, 8]. 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 secure computation protocols: the secret- sharing approach (followed by [2, 8, 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 typically have low bandwidth and send small messages per gate, in contrast to garbled circuits that are large and costly in bandwidth. This limits the potential throughput of protocols following the garbled-circuit approach, even when run in very low-latency networks. Note that information-theoretic protocols typically rely on simple operations making them fast (but no constant-round protocol for this setting is known to exist and this is conjectured to be hard if not impossible [14]).

Another question determining the type of protocol to use is 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.

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

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978331

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.

1.2 Our Results

In this paper, we focus on the question of achieving secure computation with *very high throughput* on a *fast network* (without utilizing special-purpose hardware beyond the widespread Intel AES-NI and AVX instruction sets). The challenge in achieving this is both on the computational and network levels. Specifically, a protocol achieving very high throughput would need to both be *very simple computationally* and also utilize *very little bandwidth*. Achieving both of these tasks is challenging. The BGW protocol [2] for semi-honest adversaries requires each party to send 3 field elements to each other party per multiplication gate, whereas the Sharemind protocol [4, 5, 27] (in its latest optimized version [16]) requires each party to send 5 elements per multiplication gate, and further reduces the number of field multiplications. For Boolean circuits and 3 parties, this translates to sending just 12 bits per AND gate in BGW and sending just 10 bits per gate in Sharemind.

A new protocol. We describe a new three-party protocol that is both extremely simple and has seemingly optimal bandwidth. Our protocol is suitable for arithmetic circuits over any field or over the ring modulo 2^n. Addition gates require local addition only, and multiplication gates require that each party send just a *single* field/ring element to one other party. In the Boolean case, this means that each party transmits a *single bit* only per AND gate.[1] Furthermore, the computation in our protocol is extraordinarily simple: in the case of Boolean circuits, each party carries out a single XOR operation per XOR gate, and 2 AND and 3 XOR operations per AND gate. Since all operations are merely XOR and AND, this also lends itself to parallelization on standard computers (in particular, XOR and AND over 128 bit registers can be carried out in the same time as for a single bit using Intel intrinsics).

Security. We prove that our protocol is secure in the presence of semi-honest adversaries with at most one corrupted party, under the standard simulation-based definitions. The basis of our protocol is *information theoretic* (and in fact perfectly secure). However, we save on communication by generating correlated randomness computationally, and therefore our overall protocol is computationally secure. (This combination enables us to achieve simple operations and save on additional bandwidth.) In addition to the above, we also consider a client/server model where any number of clients send shares of their inputs to 3 servers that carry out the computation for the clients and return the results to them (without learning anything). This model makes sense for "outsources secure computation services" and indeed is the business model of Cybernetica. We show that in this model, our protocol actually achieves *privacy in the presence of malicious adversaries*, meaning that a single malicious server cannot learn anything about the input or output. (We stress that this notion is strictly weaker than simulation-based security in the presence of malicious adversaries, and in particular, does not guarantee correctness. Nevertheless, it does guarantee that privacy is not breached even if one of the servers behaves maliciously.)

Number of parties. As in Sharemind [4, 5], our protocol is specifically designed for 3 parties with at most one corrupted. This is unlike BGW [2] that works for any number of parties with an honest majority. An important open question left by this paper is the design of a protocol with comparable complexity that works for any number of parties. This seems to be very challenging, based on attempts that we have made to extend our protocol.

Experimental results. We implemented our new protocol for Boolean circuits in C++ and using standard optimizations. In order to take advantage of the very simple operations required in our protocol, we used Intel intrinsics in order to carry out many executions in parallel. This is described in detail in Section 5.1. We ran our experiments on a cluster of three nodes, each with two 10-core Intel Xeon (E5-2650 v3) processors and 128GB RAM, connected via a 10Gbps Ethernet. (We remark that little RAM was utilized and thus this is not a parameter of importance here.) We carried out two main experiments, both based on securely computing the AES circuit on shared keys.

First, we computed AES in counter mode, with the aim of obtaining maximal throughput. Using the full power of the cluster (all cores), we computed over 1.3 million AES operations per second. Furthermore, utilizing a single core we achieved 100,000 AES operations per second, and utilizing 10 cores we achieved amost 1 million AES operations per second. As we will show below in Section 1.3, this way outperforms all previous protocols of this type.

Second, we wished to demonstrate that this type of protocol can be incorporated into a real system. We chose to integrate our protocol into a Kerberos KDC in order to carry out Ticket-Granting-Ticket encryption without any single server holding the encryption key (whether it be a server's key or user's hashed password). Such an architecture protects against administrators stealing passwords, or an attacker who breaches the network being able to steal all users' passwords. (We stress that in Kerberos, the raw password is never used so once the hashed password is stolen the attacker can impersonate the user.) We obtained a latency of 110ms on the server and 232ms on the client (over a LAN) for the *entire Kerberos login* (excluding database lookup). Given that this is for the purpose of user authentication, this is well within the acceptable range. In addition, we are able to support a login storm of over 35,000 user authentications per second, which is sufficient even for very large organizations.

Our results demonstrate that secure computation can be used to solve large-scale problems in practice (at least, for the cases that semi-honest security or privacy for a malicious adversary suffices).

1.3 Related Work

We compare our results with previously reported results on secure AES computation for 3 parties with an honest majority and semi-honest adversaries; see Table 1.

We stress that this table gives only very partial information since different hardware was used for each; we provide it to show the progress made and where we fit into it. However, fortunately, the setup used by us is *almost the same* as that of the latest Sharemind results in [22] (using op-

[1] This is "seemingly" optimal in terms of bandwidth, but this has not been proven and seems hard to do so; see [14].

timized code that was completely rewritten), and we now provide an in-depth comparison to it. The benchmarking in [22] was carried out between three computers with two 8-core Intel Xeon (E5-2640 v3) processors and 128GB RAM, connnected via a 10Gbps Ethernet (this configuration is described in [16] and by personal communication is that used in [22]), which is almost identical to our configuration described above. The number that we provide in Table 1 for this work is when utilizing 16 cores, and thus this is an almost identical configuration as Sharemind [22] (with 20 cores we achieve 1,324,117 AES operations per second). Observe that our latency (response time) is 70% of [23] and we achieve a throughput that is *14 times faster* than [22] (and so over an order of magnitude improvement). In fact, using a single core and a 1Gbps connection, we achieve approximately 100,000 AES operations per second (and latency of only 129ms); thus we can outperform the best Sharemind results on a very basic setup.

Year	Ref.	Latency	Throughput
2010	[10]	2000s	-
2012	[18]	14.28ms	320
2013	[19]	323ms	3450
2016	[23, Table 5.3]	223ms	25,000
2016	[22]	-	90,000
2016	this work	166ms	1,242,310

Table 1: Reported times for semi-honest 3-party computation & honest majority; the throughput is measured in **AES computations per second** (the last two rows with similar configurations).

We remark that other work on garbled circuits (e.g., two-party Yao with semi-honest adversaries) achieves much lower latency (e.g., 16ms reported in [13]). However, each garbled AES circuit is of size at least 1.3Mb (using the latest half-gates optimization [26]), not taking into account additional messages that are sent. It is therefore physically impossible to go beyond 7500 AES computations per second on a 10Gbps network (where we achieve 1.4 million). In addition, the two-party GMW approach using efficient oblivious transfer (OT) extensions is blocked by the speed of the OTs (with two OTs required per gate). Considering the communication bottleneck, each OT requires transmitting a minimum of 128 bits. Thus, the communication is approximately the same as with a garbled circuit. (The fastest known implementation [15] can process 5 million OTs per second on a 1Gbps network giving under 500 AES computations per second. This is not far from optimal assuming linear scale-up on a 10Gbps network.) Of course, we require an additional server, in contrast to the Yao and GMW protocols.

2. THE NEW PROTOCOL

In this section, we describe our new protocol for three parties. Our protocol works for arithmetic circuits over the ring modulo 2^n with Boolean circuits being a special case (with $n = 1$). The protocol uses only very simple ring addition and multiplication operations, which in the Boolean case reduces simply to bitwise AND and XOR. In addition, the protocol has very low communication: a single ring element is sent per multiplication gate and there is no communication for addition gates. In the Boolean case, we therefore have that the only communication is a single bit per AND gate.

Correlated randomness. Our protocol assumes that for every multiplication gate the three parties P_1, P_2, P_3 are given *correlated randomness* in the form of random ring elements x_1, x_2, x_3 under the constraint that $x_1 + x_2 + x_3 = 0$. We show how this can be achieved in practice with great efficiency using AES. (Thus, our protocol is information-theoretically secure with perfect correlated randomness, but the actual implementation is computationally secure due to the use of AES to generate the correlated randomness.)

2.1 Securely Computing Boolean Circuits

In order to simplify the exposition, we begin by describing the protocol for the special case of Boolean circuits with AND and XOR gates. We assume that the parties P_1, P_2, P_3 are able to obtain random $x_1, x_2, x_3 \in \{0,1\}$ such that $x_1 \oplus x_2 \oplus x_3 = 0$.

Secret sharing. We define a 2-out-of-3 secret sharing scheme, denoted $\binom{3}{2}$-sharing, as follows. In order to share a bit v, the dealer chooses three random bits $x_1, x_2, x_3 \in \{0,1\}$ under the constraint that $x_1 \oplus x_2 \oplus x_3 = 0$. Then:

- P_1's share is the pair (x_1, a_1) where $a_1 = x_3 \oplus v$.
- P_2's share is the pair (x_2, a_2) where $a_2 = x_1 \oplus v$.
- P_3's share is the pair (x_3, a_3) and $a_3 = x_2 \oplus v$.

It is clear that no single party's share reveals anything about v. In addition, any two shares suffice to obtain v; e.g., given x_1, x_2, a_1, a_2 we can compute $v = a_2 \oplus x_1$.

XOR (addition) gates. Let $(x_1, a_1), (x_2, a_2), (x_3, a_3)$ be a secret sharing of v_1, and let $(y_1, b_1), (y_2, b_2), (y_3, b_3)$ be a secret sharing of v_2. Then, in order to compute a secret sharing of $v_1 \oplus v_2$, each P_i locally computes (z_i, c_i) with $z_i = x_i \oplus y_i$ and $c_i = a_i \oplus b_i$ (no communication is needed).

In order to see that the result constitutes a valid $\binom{3}{2}$-sharing of $v_1 \oplus v_2$, observe first that $z_1 \oplus z_2 \oplus z_3 = 0$ (since both $x_1 \oplus x_2 \oplus x_3 = 0$ and $y_1 \oplus y_2 \oplus y_3 = 0$). Next, observe that for every $i \in \{1,2,3\}$ it holds that $c_i = z_{i-1} \oplus (v_1 \oplus v_2)$ where $i-1 = 3$ when $i = 1$; e.g., we have $c_1 = a_1 \oplus b_1 = x_3 \oplus v_1 \oplus y_3 \oplus v_2 = (x_3 \oplus y_3) \oplus (v_1 \oplus v_2) = z_3 \oplus (v_1 \oplus v_2)$. Thus, this constitutes a sharing of $v_1 \oplus v_2$ with randomness z_1, z_2, z_3.

AND (multiplication) gates. We now show how the parties can compute AND (equivalently, multiplication) gates; this subprotocol requires each party to send a single bit only. The protocol works in two phases: in the first phase the parties compute a simple $\binom{3}{3}$ XOR-sharing of the AND of the input bits, and in the second phase they convert the $\binom{3}{3}$-sharing into the above-defined $\binom{3}{2}$-sharing.

Let $(x_1, a_1), (x_2, a_2), (x_3, a_3)$ be a secret sharing of v_1, and let $(y_1, b_1), (y_2, b_2), (y_3, b_3)$ be a secret sharing of v_2. We assume that the parties P_1, P_2, P_3 hold *correlated randomness* α, β, γ, respectively, where $\alpha \oplus \beta \oplus \gamma = 0$. The parties compute $\binom{3}{2}$-shares of $v_1 \cdot v_2 = v_1 \wedge v_2$ as follows (from here on, we will denote multiplication of a and b by simply ab):

1. **Step 1 – compute $\binom{3}{3}$-sharing:**
   (a) P_1 computes $r_1 = x_1 y_1 \oplus a_1 b_1 \oplus \alpha$, and sends r_1 to P_2.
   (b) P_2 computes $r_2 = x_2 y_2 \oplus a_2 b_2 \oplus \beta$, and sends r_2 to P_3.
   (c) P_3 computes $r_3 = x_3 y_3 \oplus a_3 b_3 \oplus \gamma$, and sends r_3 to P_1.

   These messages are computed and sent in parallel.

2. **Step 2 – compute $\binom{3}{2}$-sharing:** In this step, the parties construct a $\binom{3}{2}$-sharing from their given $\binom{3}{3}$-sharing and the messages sent in the previous step. This requires local computation only.

   (a) P_1 stores (z_1, c_1) where $z_1 = r_1 \oplus r_3$ and $c_1 = r_1$.

   (b) P_2 stores (z_2, c_2) where $z_2 = r_2 \oplus r_1$ and $c_2 = r_2$.

   (c) P_3 stores (z_3, c_3) where $z_3 = r_3 \oplus r_2$ and $c_3 = r_3$.

Explanation of Step 1: We now show that r_1, r_2, r_3 defined in Step 1 are indeed a $\binom{3}{3}$ sharing of $v_1 v_2$, meaning that $r_1 \oplus r_2 \oplus r_3 = v_1 \wedge v_2$. Observe first that:

$$a_1 b_1 = (x_3 \oplus v_1)(y_3 \oplus v_2) = x_3 y_3 \oplus x_3 v_2 \oplus y_3 v_1 \oplus v_1 v_2 \quad (1)$$

and similarly $a_2 b_2 = x_1 y_1 \oplus x_1 v_2 \oplus y_1 v_1 \oplus v_1 v_2$, and $a_3 b_3 = x_2 y_2 \oplus x_2 v_2 \oplus y_2 v_1 \oplus v_1 v_2$. Thus,

$$\begin{aligned}
& r_1 \oplus r_2 \oplus r_3 \\
&= (x_1 y_1 \oplus a_1 b_1 \oplus \alpha) \oplus (x_2 y_2 \oplus a_2 b_2 \oplus \beta) \oplus (x_3 y_3 \oplus a_3 b_3 \oplus \gamma) \\
&= x_1 y_1 \oplus x_2 y_2 \oplus x_3 y_3 \oplus a_1 b_1 \oplus a_2 b_2 \oplus a_3 b_3 \\
&= x_1 y_1 \oplus x_2 y_2 \oplus x_3 y_3 \oplus (x_3 y_3 \oplus x_3 v_2 \oplus y_3 v_1 \oplus v_1 v_2) \\
&\qquad \oplus (x_1 y_1 \oplus x_1 v_2 \oplus y_1 v_1 \oplus v_1 v_2) \\
&\qquad \oplus (x_2 y_2 \oplus x_2 v_2 \oplus y_2 v_1 \oplus v_1 v_2) \\
&= (x_1 \oplus x_2 \oplus x_3) v_2 \oplus (y_1 \oplus y_2 \oplus y_3) v_1 \oplus v_1 v_2 = v_1 v_2
\end{aligned}$$

where the second equality is because $\alpha \oplus \beta \oplus \gamma = 0$, the third equality is from the equivalences of $a_1 b_1, a_2 b_2, a_3 b_3$ above (see Eq. (1)), the fourth equality is by cancelling repeated values and rearranging the remainder, and the last equality is because $x_1 \oplus x_2 \oplus x_3 = y_1 \oplus y_2 \oplus y_3 = 0$.

Explanation of Step 2: In order to show that the result is a valid $\binom{3}{2}$-sharing of $v_1 v_2$ according to our definition, we need to show that z_1, z_2, z_3 are such that $z_1 \oplus z_2 \oplus z_3 = 0$, and that c_1, c_2, c_3 are of the defined form.

First, $z_1 \oplus z_2 \oplus z_3 = (r_1 \oplus r_3) \oplus (r_2 \oplus r_1) \oplus (r_3 \oplus r_2) = 0$. Second, observe that since $c_1 \oplus c_2 \oplus c_3 = r_1 \oplus r_2 \oplus r_3 = v_1 v_2$ (as shown above), it holds that $c_1 = r_1 = v_1 v_2 \oplus r_2 \oplus r_3$. However, $r_2 \oplus r_3 = z_3$ (by the protocol definition) and thus $c_1 = v_1 v_2 \oplus z_3$, as required. A similar calculation shows the equality for c_2 and c_3 as well.

The above explanation shows that the gate computation "works" in the sense that the invariant of the format of the shares is preserved after every gate is computed. The fact that the protocol is secure is proved later in Section 3.

The protocol. The full 3-party protocol works in the natural way. The parties first share their inputs using the secret sharing method. They then compute each XOR and AND gate in the circuit according to a predetermined topological ordering fo the circuit. Finally, the parties reconstruct their output on the output wires. (In the client/server model, external clients send the three parties sharings of their input according, and the three parties then compute the circuit in the same way on the shares received.)

Observe that each party communicates with exactly one other party only. This property also holds for the protocol of Sharemind [4, 5]. However, our secret-sharing scheme and multiplication protocol are completely different.

2.2 Generating Correlated Randomness

Our protocol relies on the fact that the parties hold random bits $\alpha, \beta, \gamma \in \{0,1\}$ such that $\alpha \oplus \beta \oplus \gamma = 0$ for every AND gate. In this section, we show how the parties can efficiently generate such α, β, γ.

Information-theoretic correlated randomness. It is possible to securely generate correlated randomness with perfect security by having each party P_i simply choose a random $\rho_i \in \{0,1\}$ and send it to P_{i+1} (where P_3 sends to P_1). Then, each party takes its random bit to be the XOR of the bit it chose and the bit it received: P_1 computes $\alpha = \rho_3 \oplus \rho_1$, P_2 computes $\beta = \rho_1 \oplus \rho_2$ and $\gamma = \rho_2 \oplus \rho_3$. Observe that $\alpha + \beta + \gamma = 0$ as required. In addition, if P_1 is corrupted, then it knows nothing about β and γ except that $\beta \oplus \gamma = \alpha$. This is because β and γ both include ρ_2 in their computation and this is unknown to P_1. A similar argument holds for a corrupted P_2 or P_3. Despite the elegance and simplicity of this solution, we use a different approach. This is due to the fact that this would *double* the communication per AND gate; it is true that this is still very little communication. However, given that communication is the bottleneck, it would halve the throughput.

Computational correlated randomness. We now show how it is possible to securely compute correlated randomness *computationally* without *any interaction* beyond a short initial setup. This enables us to maintain the current situation where parties need only transmit a single bit per AND gate. This method is similar to that of the PRSS subprotocol in [9], but simpler since Shamir sharing is not needed. Let κ be the security parameter, and let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}$ be a pseudorandom function outputting a single bit.

1. **Init:**

   (a) Each P_i chooses a random $k_i \in \{0,1\}^\kappa$.

   (b) Party P_1 sends k_1 to P_3, party P_2 sends k_2 to P_1 and party P_3 sends k_3 to k_1.

   P_1 holds k_1, k_2, P_2 holds k_2, k_3 and P_3 holds k_3, k_1.

2. **GetNextBit:** Given a unique identifier $id \in \{0,1\}^\kappa$,

   (a) P_1 computes $\alpha = F_{k_1}(id) \oplus F_{k_2}(id)$.

   (b) P_2 computes $\beta = F_{k_2}(id) \oplus F_{k_3}(id)$.

   (c) P_3 computes $\gamma = F_{k_3}(id) \oplus F_{k_1}(id)$.

Observe that $\alpha \oplus \beta \oplus \gamma = 0$. Furthermore, P_1 does not know k_3 which is used to generate β and γ. Thus, β and γ are pseudorandom to P_1, under the constraint that $\beta \oplus \gamma = \alpha$. In practice, the id can be a counter that all parties locally increment at every call to **GetNextBit**.

2.3 The Ring Modulo 2^n and Fields

Our protocol above works for Boolean circuits. However, in some cases arithmetic circuits are far more efficient. In this section, we show how to generalize the protocol above to the general case of the ring modulo 2^n and arbitrary fields of size greater than 2. We describe the protocol for the ring modulo 2^n; it is clear that everything holds for arbitrary finite fields and rings in which 3^{-1} exists. (The only thing that is needed is to be able to divide by 3 which is defined by adding the unity to itself 3 times. This is possible in the ring modulo 2^n since $\gcd(3, 2^n) = 1$, and is always possible in a field.) From here-on in this section, all arithmetic is modulo 2^n.

We remark that when taking $n = 1$ we have that addition (and subtraction) is the same as XOR, and multiplication is the same as AND. In this case, the protocol here is exactly that described in Section 2.1.

$\binom{3}{2}$-secret sharing. In order to share an element v mod 2^n the dealer chooses three random elements $x_1, x_2, x_3 \in \mathbb{Z}_{2^n}$

under the constraint that $x_1+x_2+x_3=0$. Then, P_1's share is (x_1,a_1) where $a_1=x_3-v$, P_2's share is (x_2,a_2) where $a_2=x_1-v$, and P_3's share is (x_3,a_3), where $a_3=x_2-v$. As in the Boolean case, it is easy to see that each share reveals nothing of v, and that any two shares suffice to reconstruct v.

We now show that each party's share reveals nothing about the secret. For simplicity, we show this for P_1 (all others are shown in a similar way). Party P_1's share consists of the pair (a_1,x_1) where $a_1=x_3-v$. Since x_1,x_2,x_3 are random under the constraint that $x_1+x_2+x_3=0$, this is equivalent to x_1 and x_3 being chosen independently at random and then x_2 being chosen to equal $-x_1-x_3$. In this light, a_1 in P_1's share is a one-time pad encryption of v using random key x_3, and x_1 is an independent random value. Thus, P_1's share reveals nothing about v whatsoever. This implies:

LEMMA 2.1. *For any two values $v_a,v_b\in\mathbb{Z}_{2^n}$ and any $i\in\{1,2,3\}$, the distribution over P_i's share (x_i,a_i) of v_a is* identical *to the distribution over P_i's share (y_i,b_i) of v_b.*

Addition gates. As in the Boolean case, addition gates are computed by locally adding the shares modulo 2^n.

Multiplication gates: Let $(x_1,a_1),(x_2,a_2),(x_3,a_3)$ be a secret sharing of v_1, and let $(y_1,b_1),(y_2,b_2),(y_3,b_3)$ be a secret sharing of v_2, and assume the parties P_1,P_2,P_3 hold $\alpha,\beta,\gamma\in\mathbb{Z}_{2^n}$ respectively, such that $\alpha+\beta+\gamma=0$. In order to compute a $\binom{3}{2}$-sharing of the product of two values, the parties work as above with the following differences:

1. P_1 computes $r_1=\frac{a_1b_1-x_1y_1+\alpha}{3}$ and sends r_1 to P_2.
2. P_2 computes $r_2=\frac{a_2b_2-x_2y_2+\beta}{3}$ and sends r_2 to P_3.
3. P_3 computes $r_3=\frac{a_3b_3-x_3y_3+\gamma}{3}$ and sends r_3 to P_1.
4. P_1 defines its share as $z_1=r_3-r_1$ and $c_1=-2r_3-r_1$.
5. P_2 defines its share as $z_2=r_1-r_2$ and $c_2=-2r_1-r_2$.
6. P_3 defines its share as $z_3=r_2-r_3$ and $c_3=-2r_2-r_3$.

We remark that the above computation is legal since 3 is relatively prime to 2^n; thus 3 has an inverse. In addition, the above all holds in finite fields with more than 3 elements.

In order to see that $r_1+r_2+r_3=v_1v_2$, first observe that

$$a_1b_1=(x_3-v_1)(y_3-v_2)=x_3y_3-x_3v_2-y_3v_1+v_1v_2 \quad (2)$$

and likewise $a_2b_2=x_1y_1-x_1v_2-y_1v_1+v_1v_2$ and $a_3b_3=x_2y_2-x_2v_2-y_2v_1+v_1v_2$. Then,

$$\begin{aligned}3(r_1+r_2+r_3)&=a_1b_1-x_1y_1+\alpha+a_2b_2-x_2y_2+\beta+a_3b_3-x_3y_3+\gamma\\&=a_1b_1+a_2b_2+a_3b_3-x_1y_1-x_2y_2-x_3y_3\\&=3v_1v_2-v_1(y_1+y_2+y_3)-v_2(x_1+x_2+x_3)=3v_1v_2\end{aligned}$$

where the second equality holds since $\alpha+\beta+\gamma=0$ and the third equality follows by plugging in the equivalences of a_1b_1,a_2b_2,a_3b_3 above (see Eq. (2)) and rearranging the elements, and the fourth equality follows from the fact that $x_1+x_2+x_3=y_1+y_2+y_3=0$. Since we can divide by 3 in this ring (and in a field) we have that $r_1+r_2+r_3=v_1v_2$.

Next, we show that the shares the parties hold are a valid $\binom{3}{2}$-sharing of v_1v_2 according to our definition. i.e, that the shares are $(z_1,z_3-v_1v_2),(z_2,z_1-v_1v_2)$ and $(z_3,z_2-v_1v_2)$ such that $z_1+z_2+z_3=0 \bmod 2^n$. First, observe that the sum of the first elements of the shares is $z_1+z_2+z_3=(r_3-r_1)+(r_1-r_2)+(r_2-r_3)=0$ as required. Second, for party P_1 it holds that, $c_1=-2r_3-r_1=-r_3-r_3-r_1-r_2+r_2=(r_2-r_3)-(r_1+r_2+r_3)=z_3-v_1v_2$ as required (recall that $z_3=r_2-r_3$ in the protocol). Correctness for P_2 and P_3 follows similarly.

In the proof of security, we show that the secret is perfectly hidden by the resulting secret sharing. This hiding follows from the use of the correlated randomness in the computation. In particular, the random values α,β,γ that the parties add in the local computation perfectly mask the value on the wire.

Generating correlated randomness. The parties use the same (computational) method as described in Section 2.2, with the following differences. First, we assume that F_k is a pseudorandom function mapping strings into $\mathbb{Z}_{2^n}$ (or equivalently to $\{0,1\}^n$). Second, party P_1 computes $\alpha=F_{k_1}(id)-F_{k_2}(id)$, party P_2 computes $\beta=F_{k_2}(id)-F_{k_3}(id)$, and party P_3 computes $\gamma=F_{k_3}(id)-F_{k_1}(id)$.

2.4 Protocol Efficiency and Comparison

In the case of arbitrary finite fields, Shamir's secret-sharing [24] is "ideal", meaning that the size of the share equals the size of the secret (which is minimum size), as long as the number of parties is less than the size of the field. In our protocol, the secret sharing scheme is not ideal since it consists of two ring or field elements instead of a single field element. However, this is of little consequence when considering the efficiency of the protocol since our protocol requires only sending a *single* element per multiplication gate. In addition, the computation consists merely of two multiplications and two additions.

In comparison, the BGW protocol [2, 1] requires transmitting two field elements per multiplication gate by each party when using [21] method (with a single round of communication). In addition, when considering Boolean circuits, at least two bits are needed per field element, since there are 3 parties. Furthermore, the computation requires polynomial evaluations which are far more expensive.

In the Sharemind protocol [4, 5], the parties transmit five ring elements per AND gate over two communication rounds, and compute 3 multiplications and 8 additions. We remark that our method for generating correlated randomness can be used to reduce the number of elements sent in the Sharemind protocol from 5 to 2 and to reduce the number of communication rounds to 1.

3. SECURITY – SEMI-HONEST ADVERSARIES

In this section, we prove that our protocol is secure in the presence of one semi-honest adversarial party (in Section 4 we prove that the protocol is private in the presence of one malicious adversary). Semi-honest security is sufficient when parties somewhat trust each other, but are concerned with inadvertent leakage or cannot share their raw information due to privacy regulations. It is also sufficient in cases where it is reasonable to assume that the parties running the protocol are unable to replace the installed code. Nevertheless, security against covert or malicious adversaries is preferable, providing far higher guarantees; we leave extensions of our protocol to these settings for future work.

Since the protocol for Boolean circuits is a *special case* of the protocol for the ring modulo 2^n, we prove the security for the case of the ring modulo 2^n. The proof is identical in

the case of fields with more than 3 elements. Throughout, in order to simplify notation, when we use an index (say, i) to denote the ith party (with $i \in \{1,2,3\}$), we will write $i-1$ and $i+1$ to mean the "previous" and "subsequent" party, respectively. That is, when $i=1$ then $i-1=3$ and when $i=3$ then $i+1=1$.

3.1 Preliminaries

We use the definition of security in the presence of semi-honest adversaries as in [6, 11], making the necessary changes to formalize perfect security as well.

Perfect security in the presence of semi-honest adversaries. Loosely speaking, a protocol is secure in the presence of one corrupted party if the view of the corrupted party in a real protocol execution can be generated by a simulator given only the corrupted party's input and output. The view of party i during an execution of a protocol π on inputs $\vec{x}$, denoted $\text{VIEW}_i^\pi(\vec{x})$, consists of its input x_i, its internal random coins r_i and the messages that were received by i in the execution. The output of all parties from an execution of π is denoted by $\text{OUTPUT}^\pi(\vec{x})$.

DEFINITION 3.1. *Let $f:(\{0,1\}^*)^3 \to (\{0,1\}^*)^3$ be a* probabilistic 3*-ary functionality and let π be a protocol. We say that π* computes f with perfect security in the presence of one semi-honest corrupted party *for f if there exists a probabilistic polynomial-time algorithm $\mathcal{S}$ such that for every corrupted party $i \in \{1,2,3\}$, and every $\vec{x} \in (\{0,1\}^*)^3$ where $|x_1|=|x_2|=|x_3|$:*

$$\left\{ (\mathcal{S}(x_i, f_i(\vec{x})), f(\vec{x})) \right\} \equiv \left\{ (\text{VIEW}_i^\pi(\vec{x}), \text{OUTPUT}^\pi(\vec{x})) \right\} \quad (3)$$

If Eq. (3) holds with computational indistinguishability, then we say that π computes f with computational security in the presence of one semi-honest corrupted party.

The above definition is for the general case of probabilistic functionalities, where we consider the joint distribution of the output of $\mathcal{S}$ and of the parties. For the case of deterministic functionalities, however, we can separate the correctness and privacy requirements, and use a simpler and easier to prove definition. As shown in [11](see section 7.3.1), *any probabilistic functionality* can be privately computed in the presence of t corrupted parties using a general protocol which computes *any deterministic functionality* in the presence of t corrupted parties. Therefore, in order to prove the security of our protocol we can use the definition for deterministic functionalities stated below.

DEFINITION 3.2. *Let $f:(\{0,1\}^*)^3 \to (\{0,1\}^*)^3$ be a* deterministic 3*-ary functionality and let π be a protocol. We say that π* computes f with perfect security in the presence of one semi-honest corrupted party *for f, if for every $\vec{x} \in (\{0,1\}^*)^3$ where $|x_1|=|x_2|=|x_3|$, it holds that $\text{OUTPUT}^\pi(\vec{x}) = f(\vec{x})$, and there exists a probabilistic polynomial-time algorithm $\mathcal{S}$ such that for every corrupted party $i \in \{1,2,3\}$, and every $\vec{x} \in (\{0,1\}^*)^3$ where $|x_1|=|x_2|=|x_3|$:*

$$\{\mathcal{S}(x_i, f_i(\vec{x}))\} \equiv \{\text{VIEW}_i^\pi(\vec{x})\}$$

We prove the security of our protocols using the hybrid model, where parties run a protocol with real messages and also have access to a trusted party computing a subfunctionality for them. The modular sequential composition theorem of [7] states that replacing the trusted party computing the subfunctionality with a real secure protocol results in the same output distribution. When the subfunctionality is g, we say that the protocol works in the g-hybrid model.

Universal composability. Protocols that are proven secure in the universal composability framework [7] have the property that they maintain their security when run in parallel and concurrently with other secure and insecure protocols. In [17, Theorem 1.5], it was shown that any protocol that is proven secure with a black-box non-rewinding simulator and also has the property that the inputs of all parties are fixed before the execution begins (called input availability or start synchronization in [17]), is also secure under universal composability. Since the input availability property holds for all of our protocols and subprotocols, it is sufficient to prove security in the classic stand-alone setting and automatically derive universal composability from [17]. We remark that this also enables us to call the protocol and subprotocols that we use in parallel and concurrently (and not just sequentially), enabling us to achieve more efficient computation (e.g., by running many executions in parallel or by running each layer of a circuit in parallel).

Proof outline. We denote a protocol π in the g-hybrid model by π^g, and the real protocol obtained by replacing calls to g by invocations of subprotocol ρ by π^ρ. We abuse notation and write $\pi^g \equiv f$ to say that π securely computes f in the g-hybrid model, and write $\pi^\rho \equiv f$ to say that the real protocol π^ρ securely computes f. Denote by σ the protocol that computes the correlated randomness functionality $\mathcal{F}_{\text{CR}}$, by ρ the protocol that computes the multiplication functionality $\mathcal{F}_{\text{MULT}}$ in the $\mathcal{F}_{\text{CR}}$-hybrid model, and by π the protocol that computes the functionality f in the $\mathcal{F}_{\text{MULT}}$-hybrid model. Our goal is to prove that π^{ρ^σ} securely computes f in the presence of one static semi-honest corrupted party.

Let f be a 3-ary functionality. We begin by proving that $\pi^{\mathcal{F}_{\text{MULT}}}$ computes f with *perfect* security in the presence of one static semi-honest party. Next, we prove that $\rho^{\mathcal{F}_{\text{CR}}}$ computes $\mathcal{F}_{\text{MULT}}$ with *perfect* security in the presence of one static semi-honest party in the $\mathcal{F}_{\text{CR}}$-hybrid model. Finally, we prove that σ computes $\mathcal{F}_{\text{CR}}$ with *computational* security in the presence of one static semi-honest party. The reason for achieving only computational security for the correlated randomness protocol is that we use a pseudorandom function to compute the random values. The proof in this case, thereby, works by making a reduction to a distinguisher between a pseudorandom function and a random function.

Once we have proved that $f \equiv \pi^{\mathcal{F}_{\text{MULT}}}$, that $\mathcal{F}_{\text{MULT}} \equiv \rho^{\mathcal{F}_{\text{CR}}}$ and that $\mathcal{F}_{\text{CR}} \stackrel{\text{c}}{\equiv} \sigma$, we can apply the composition theorem of [7] (using the fact that universal composability is implied via [17]) to conclude that $\pi^{\rho^{\mathcal{F}_{\text{CR}}}} \equiv f$; that is, π^{ρ^σ} computes f with computationl security in the presence of one static semi-honest adversary.

3.2 Computing f in the $\mathcal{F}_{\text{MULT}}$-Hybrid Model

We define the multiplication functionality $\mathcal{F}_{\text{MULT}}$ that receives input shares of two values v_a, v_b as input and outputs shares of the product $v_a v_b$, according to the secret-sharing scheme described in Section 2.3. Intuitively, $\mathcal{F}_{\text{MULT}}$ should be defined by receiving the shares of all parties, reconstructing the values v_1, v_2 from the shares, and then generating a random resharing of the $v_1 v_2$. Indeed, if secure coin tossing were used instead of the method that we use for correlated randomness, then $\mathcal{F}_{\text{MULT}}$ would be defined in this natural

way. However, this would require additional communication and would affect performance. We therefore need to define a more complex multiplication functionality. In order to understand why this is needed, recall the real protocol and consider the specific case that P_1 is corrupted. In order to simplify this explanation, consider the Boolean case.

Party P_1 computes $r_1 = x_1y_1 \oplus a_1b_1 \oplus \alpha$ and receives r_3 from P_3. Observe that α is not random to the corrupted P_1 and is fixed by a very specific computation (specifically, $F_{k_1}(id) \oplus F_{k_2}(id)$; see Section 2.2). Thus, P_1's computation of r_1 is *deterministic*. Now, P_1's output from the multiplication protocol is the pair (z_1, c_1) where $z_1 = r_1 \oplus r_3$ and $c_1 = r_1$. Since r_3 is received from P_3 and is masked with the correlated randomness that P_3 receives (which is generated using a pseudorandom function with a key not known to P_1) this value is random. However, c_1 is *fixed* (since it equals r_1). Stated differently, given that r_1 is fixed, there are exactly two possible values for (z_1, c_1) based on $z_1 = 0$ or $z_1 = 1$. In contrast, a random secret sharing has four possible values for (z_1, c_1), with all four combinations of $z_1, c_1 \in \{0, 1\}$. Thus, it is *not* true that the multiplication protocol generates a new random sharing of the product.

In order to solve this problem, we take a different approach. We allow the corrupted party to completely determine its share (z_i, c_i). The functionality $\mathcal{F}_{\text{MULT}}$ then determines the other parties' shares based on (z_i, c_i) and the product $v_a v_b$. Interestingly, in this secret sharing method, a single share together with the secret fully determines all other shares. This is because each $c_i = z_{i-1} - v_a v_b$. Thus, (z_i, c_i) and $v_a v_b$ determines $z_{i-1} = c_i + v_a v_b$, which in turn determines z_{i+1} since $z_1 + z_2 + z_3 = 0$. Finally, all z values together with $v_a v_b$ determine all c values. Formally:

FUNCTIONALITY 3.3 ($\mathcal{F}_{\text{MULT}}$ – MULTIPLICATION).

1. $\mathcal{F}_{\text{MULT}}$ receives $((x_j, a_j), (y_j, b_j))$ from each P_j and receives a pair $(z_i, c_i) \in \mathbb{Z}_{2^n} \times \mathbb{Z}_{2^n}$ from the adversary controlling P_i.
2. $\mathcal{F}_{\text{MULT}}$ computes $v_a = x_1 - a_2$ and $v_b = y_1 - b_2$ and $v_c = v_a v_b$.
3. $\mathcal{F}_{\text{MULT}}$ sets $z_{i-1} = c_i + v_c$ and $z_{i+1} = -z_i - z_{i-1}$, and sets $c_{i-1} = z_{i+1} - v_c$ and $c_{i+1} = z_i - v_c$.
4. $\mathcal{F}_{\text{MULT}}$ sends each P_j the pair (z_j, c_j) (for $j \in \{1, 2, 3\}$).

We denote the protocol for securely computing f that is defined in Section 2.3 by Protocol 2.3. We now prove the security of Protocol 2.3 according to Definition 3.2.

THEOREM 3.4. *Let $f : ((\mathbb{Z}_{2^n})^*)^3 \to ((\mathbb{Z}_{2^n})^*)^3$ be a 3-ary functionality. Then, Protocol 2.3 computes f with perfect security in the $\mathcal{F}_{\text{MULT}}$-hybrid model, in the presence of one semi-honest corrupted party.*

PROOF SKETCH: Since the circuit C computes functionality f the first (correctness) requirement of Definition 3.2 is immediately fulfilled. We now proceed to the second (privacy) requirement of the definition. Intuitively, the protocol is private since the corrupted party receives *nothing* in the execution beyond shares on the input wires which are uniformly distributed and the shares on the output wires. In particular, addition gates consist of local computation only, and multiplication gates are computed using the $\mathcal{F}_{\text{MULT}}$ functionality. However, in $\mathcal{F}_{\text{MULT}}$, the adversary defines the corrupted party's share (z_i, c_i) as it likes and receives nothing back (formally, it receives back (z_i, c_i) but this is already known). Thus, this can reveal nothing whatsoever about the actual values on the wires in the computation. Finally, for each output wires in which P_i receives output, given its share (z_i, c_i) on the output wire and given the real output value v, the simulator can generate the exact shares that P_i would receive from the other parties. This is due to the fact mentioned above that a single share plus the actual secret fully determines the other two shares (and can be computed efficiently in the same way as the functionality). It follows that we can construct a simulator that simply defines the view of the corrupted party to be shares of arbitrary values for the input wires, and provide the shares received on the output wires (after running the adversary and receiving the shares it chooses for its output wires). By Lemma 2.1, the simulator-generated view of the corrupted party is identically distributed to that of a real execution. ■

3.3 Computing $\mathcal{F}_{\text{MULT}}$ in the $\mathcal{F}_{\text{CR}}$-Hybrid Model

In this section, we prove that the multiplication protocol described in Section 2.3 computes the $\mathcal{F}_{\text{MULT}}$ functionality with perfect security in the presence of one semi-honest corrupted party. Recall that our protocol utilizes correlated randomness in the form of random $\alpha_1, \alpha_2, \alpha_3$ such that $\alpha_1 + \alpha_2 + \alpha_3 = 0$.

Background – correlated randomness. First, we formally define the ideal functionality $\mathcal{F}_{\text{CR}}$. A naive definition would be to have the ideal functionality choose $\alpha_1, \alpha_2, \alpha_3$ and send α_i to P_i for $i \in \{1, 2, 3\}$. However, securely realizing such a functionality would require interaction (as in the information-theoretic method first described in Section 2.2). In order to model our computational method described in Section 2.2 (which is the same as used for the ring case) we need to take into account that the corrupted party's value is generated in a very specific way using a pseudorandom function. In order for the $\mathcal{F}_{\text{MULT}}$ protocol to be secure, all that is needed is that the corrupted party knows *nothing* about the honest party's values (beyond the given constraint that all values sum to zero). In particular, there is no requirement regarding how the corrupted party's value is generated. Recall that in our protocol each party holds two keys which are used to locally compute the correlated randomness. In order for the view of the corrupted party to be like in the real protocol, we define the functionality $\mathcal{F}_{\text{CR}}$ so that it generates the corrupted party's value in this exact same way (i.e., $F_k(id) - F_{k'}(id)$ for keys k, k'; see Section 2.3). As we have mentioned, the honest parties' values *are* chosen randomly, under the constraint that all values sum to zero.

The functionality is described formally in Functionality 3.5. The functionality chooses two keys k, k' for a pseudorandom function F and sends them to the corrupted party. We denote by κ the computational security parameter, and thus the length of the keys k, k'.

FUNCTIONALITY 3.5 ($\mathcal{F}_{\text{CR}}$ – CORR. RANDOMNESS).

Let $F : \{0,1\}^* \times \{0,1\}^* \to \mathbb{Z}_{2^n}$ be a keyed function. Upon invocation, $\mathcal{F}_{\text{CR}}$ chooses a pair of keys $k, k' \in \{0,1\}^\kappa$ and sends them to the adversary controlling party P_i. Then:

- Upon receiving input id from all parties, functionality $\mathcal{F}_{\text{CR}}$ computes $\alpha_i = F_k(id) - F_{k'}(id)$ and chooses random values $\alpha_{i-1}, \alpha_{i+1} \in \mathbb{Z}_{2^n}$ under the constraint that $\alpha_1 + \alpha_2 + \alpha_3 = 0 \bmod 2^n$. $\mathcal{F}_{\text{CR}}$ sends α_j to P_j for every j.

The multiplication protocol. A formal description of the protocol that securely computes the multiplication functionality $\mathcal{F}_{\text{MULT}}$ in the $\mathcal{F}_{\text{CR}}$-hybrid model appears in Protocol 3.6.

PROTOCOL 3.6 (COMPUTING $\mathcal{F}_{\text{MULT}}$).

- **Inputs**: Each party P_j (with $j \in \{1,2,3\}$) holds two pairs of values $(x_j, a_j), (y_j, b_j)$ which are valid $\binom{3}{2}$-sharings of the values that are on the input wires.
- **Auxiliary input:** The parties hold the same unique identifier id (in the protocol using $\mathcal{F}_{\text{MULT}}$ this identifier can be the index of the multiplication gate being computed).
- **The protocol**:
  1. **Correlated randomness:** Each party P_j (with $j \in \{1,2,3\}$) sends id to $\mathcal{F}_{\text{CR}}$ and receives back α_j from $\mathcal{F}_{\text{CR}}$.
  2. **Local computation**: Each party P_j locally computes: $r_j = \frac{a_j b_j - x_j y_j + \alpha_j}{3}$.
  3. **Communication**: Party P_j sends r_j to party P_{j+1} (recall that $P_{j+1} = P_1$ when $j = 3$).
- **Output**: Each P_j outputs (z_j, c_j) where $z_j = r_{j-1} - r_j$ and $c_j = -2r_{j-1} - r_j$; recall $r_{j-1} = r_3$ when $j = 1$.

We now prove that the protocol is secure in the presence of one static semi-honest corrupted party.

THEOREM 3.7. *Protocol 3.6 computes $\mathcal{F}_{\text{MULT}}$ with perfect security in the $\mathcal{F}_{\text{CR}}$-hybrid model in the presence of one semi-honest corrupted party.*

PROOF. In the protocol, the corrupted party receives a single message. This message is an element from $\mathbb{Z}_{2^n}$ which is uniformly distributed over $\mathbb{Z}_{2^n}$, due to the fact that each party masks its message using a random value received from the $\mathcal{F}_{\text{CR}}$ functionality. Intuitively, the protocol is secure because all the corrupted party sees is a random element. (Note that the corrupted party also receives output from $\mathcal{F}_{\text{CR}}$ but this is fully determined to be $\alpha_i = F_k(id) - F_{k'}(id)$.) We now prove this claim formally.

The $\mathcal{F}_{\text{MULT}}$ functionality as we have defined it is deterministic, and we therefore prove security via the simpler Definition 3.2. In order to show correctness, we need to show that the actual values $(z_1, c_1), (z_2, c_2), (z_3, c_3)$ output by all three parties from Protocol 3.6 are exactly the same values as those computed by $\mathcal{F}_{\text{MULT}}$. In order to see that this holds, recall that in Section 2.3 we showed that

$$z_1 + z_2 + z_3 = 0 \quad \text{and} \quad \forall j \in \{1,2,3\}\ c_j = z_{j-1} - v_a v_b. \quad (4)$$

We claim that given a fixed (z_i, c_i) and $v_a v_b$, Eq. (4) implies that all values $z_{i-1}, c_{i-1}, z_{i+1}, c_{i+1}$ are *fully determined*. Specifically, let (z_i, c_i) be fixed and let $v_a v_b$ be the output value. Since for all $j \in \{1,2,3\}$ we have $c_j = z_{j-1} - v_a v_b$, this implies that $z_{i-1} = c_i + v_a v_b$ is determined, which in turn determines $z_{i+1} = -z_i - z_{i-1}$. Finally, this determines $c_{i+1} = z_i - v_a v_b$ and $c_{i-1} = z_{i+1} - v_a v_b$. This is exactly the way that $\mathcal{F}_{\text{MULT}}$ computes the output values, and thus these are identical in the protocol and in the functionality output.

We now prove privacy by defining the simulator. The simulator $\mathcal{S}$ receives the input and output of the corrupted party P_i from $\mathcal{F}_{\text{MULT}}$ as well as the auxiliary input id and (k, k'), and needs to compute the messages P_i sees during the execution. The input of the corrupted party P_i consists of two pair of shares $(x_i, a_i), (y_i, b_i)$ and it has no output. Intuitively, $\mathcal{S}$ chooses a random element $r_{i-1} \in \mathbb{Z}_{2^n}$ and uses it to define the pair (z_i, c_i) that it sends to the trusted party computing $\mathcal{F}_{\text{MULT}}$. Formally, the simulator receives $(((x_i, a_i), (y_i, b_i))$ and works as follows:

1. $\mathcal{S}$ chooses a random $r_{i-1} \in \mathbb{Z}_{2^n}$.
2. $\mathcal{S}$ sets $r_i = \frac{a_i b_i - x_i y_i + \alpha_i}{3}$ where $\alpha_i = F_k(id) - F_{k'}(id)$ as would be computed by $\mathcal{F}_{\text{CR}}$ in the protocol.
3. $\mathcal{S}$ sets $z_i = r_{i-1} = r_i$ and $c_i = -2r_{i-1} - r_i$.
4. $\mathcal{S}$ sends (z_i, c_i) to $\mathcal{F}_{\text{MULT}}$.
5. $\mathcal{S}$ adds α_i and r_{i-1} to the view of the corrupted party.

The values α_i and r_i are computed by $\mathcal{S}$ exactly as by P_i in a real execution. The only difference is how r_{i-1} is computed; P_i receives $r_{i-1} = \frac{a_{i-1} b_{i-1} - x_{i-1} y_{i-1} + \alpha_{i-1}}{3}$ from P_{i-1} in a real execution, whereas $\mathcal{S}$ chooses $r_{i-1} \in \mathbb{Z}_{2^n}$ uniformly at random in the simulation. The distribution over these two values is *identical* by the fact that $\mathcal{F}_{\text{CR}}$ chooses $\alpha_{i-1}, \alpha_{i+1}$. Specificaly, $\mathcal{F}_{\text{CR}}$ chooses these at random under the constraint that $\alpha_1 + \alpha_2 + \alpha_3 = 0$. However, this is equivalent to choosing $\alpha_{i-1} \in \mathbb{Z}_{2^n}$ uniformly at random and then setting $\alpha_{i+1} = -\alpha_i - \alpha_{i-1}$. Now, since α_{i-1} is uniformly random, this implies that r_{i-1} is uniformly random (since it is independent of all other values used in the generation of r_{i-1}). Thus, the distribution over the real r_{i-1} received by P_i in the protocol execution and over the simulated r_{i-1} generated by $\mathcal{S}$ is identical. This completes the proof. ■

3.4 Computing $\mathcal{F}_{\text{CR}}$ in the Plain Model

In this section, we prove that our protocol privately computes the $\mathcal{F}_{\text{CR}}$ functionality in the presence of one semi-honest corrupted party. We have already presented the $\mathcal{F}_{\text{CR}}$ functionality in Functionality 3.5. The protocol for computing it appears in Protocol 3.8.

PROTOCOL 3.8 (COMPUTING $\mathcal{F}_{\text{CR}}$).

- **Auxiliary input**: Each party holds a security parameter κ, a description of a pseudorandom function $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \mathbb{Z}_{2^n}$.
- **Setup (executed once)**:
  1. Each party P_j chooses randomly $k_j \in \{0,1\}^\kappa$.
  2. Each party P_j sends k_j to party P_{j+1}.
- **Generating randomness**: Upon input id, each party P_j computes $\alpha_j = F_{k_j}(id) - F_{k_{j-1}}(id)$ and **outputs** it.

THEOREM 3.9. *If $F_k()$ is a pseudorandom function, then Protocol 3.8 computes $\mathcal{F}_{\text{CR}}$ with computational security in the plain model, in the presence of 1 semi-honest corrupted party.*

PROOF SKETCH: Since the functionality is probabilistic, we need to use Definition 3.1. Unlike the previous security proofs we have seen, the security of this protocol is computational and it relies on the assumption that F_k is a pseudorandom function. Thus, we will show that the ability to distinguish between the outputs in the real and ideal executions can be used to distinguish between the pseudorandom function and a truly random function, in contradiction to the assumption.

Let P_i be the corrupted party. We define the simulator $\mathcal{S}$ who simulates P_i's view. $\mathcal{S}$ is invoked on the security parameter 1^κ and works as follows:

1. $\mathcal{S}$ receives k, k' from $\mathcal{F}_{\text{CR}}$ when it is first invoked (see Functionality 3.5).

2. $\mathcal{S}$ sets the random tape of P_i (used by P_i to sample k_i) to be the key k received from $\mathcal{F}_{\text{CR}}$.

3. $\mathcal{S}$ simulates the setup phase by writing the key k' as the key k_{i-1} received by P_i from P_{i-1}.

4. From this point on, every time that P_i receives id for input, $\mathcal{S}$ sends it to the trusted party computing $\mathcal{F}_{\text{CR}}$. (P_i receives back α_i but this equals $F_k(id) - F_{k'}(id) = F_{k_i}(id) - F_{k_{i-1}}(id)$ and is known to P_i. Also, this value is computed locally by P_i in the protocol and not received. Thus, $\mathcal{S}$ does not include it in P_i's view.)

It is easy to see that the view generated by the simulator which consists of the P_i's random tape and the incoming message k_{i-1} is distributed identically to its view in a real execution. However, this is not sufficient, as we need to prove indistinguishability of the *joint distribution* of both the corrupted party's view and the honest parties' outputs. Observe that in the real protocol execution, the honest parties' outputs are generated using the pseudorandom function, whereas in the ideal world they are chosen randomly by $\mathcal{F}_{\text{CR}}$.

Intuitively, the proof follows from the fact that both P_{i-1} and P_{i+1} generate their values using the pseudorandom function F with key k_{i+1} that is independent of k_i and k_{i-1}. Thus, replacing $F_{k_{i+1}}$ with a truly random function f results in P_{i-1} and P_{i+1} generating values α_{i-1} and α_{i+1} that are random under the constraint that $\alpha_1 + \alpha_2 + \alpha_3 = 0$. (Specifically, P_{i-1} generates $\alpha_{i-1} = F_{k_{i-1}}(id) - f(id)$ and P_{i+1} generates $\alpha_{i+1} = f(id) - F_{k_i}(id)$. Thus, $\alpha_{i-1} + \alpha_{i+1} = F_{k_{i-1}}(id) - f(id) + f(id) - F_{k_i}(id) = F_{k_{i-1}}(id) - F_{k_i}(id) = -\alpha_i$, as required.) The full proof follows via a straightforward reduction. ∎

3.5 Wrapping Up

In the previous sections, we have proven that Protocol 2.3 computes any 3-ary functionality with perfect security in the F_{mult}-hybrid model, and that Protocol 3.6 computes the F_{mult} functionality with perfect security in the $\mathcal{F}_{\text{CR}}$-hybrid model. Finally, we have proved that Protocol 3.8 computes $\mathcal{F}_{\text{CR}}$ with computational security (in the plain model) under the assumption that pseudorandom functions exist. (All of the above holds for a single corrupted party in the semi-honest model.) Using the fact that all our protocols are UC secure from [17] and thus applying the UC composition theorem of [7], we conclude with the following theorem:

THEOREM 3.10. *Assume that F is a pseudorandom function, and let f be a 3-ary functionality. Then, Protocol 2.3 computes f with computational security, in the presence of one semi-honest corrupted party.*

4. PRIVACY FOR MALICIOUS ADVERSARIES IN THE CLIENT-SERVER MODEL

In this section, we consider the "client-server" model where the parties running the multiparty computation protocol are servers who receive the input shares of multiple clients and compute the output for them. This is the model used by Cybernetica in their Sharemind product [4]. In this model, the servers do not see any of the inputs nor any of the outputs. Rather, they receive *shares* of the inputs and send the clients shares of their output. Since the parties running the multiparty protocol do not have any input or output, it is possible to formulate an indistinguishability-based definition of security, saying that a corrupted server learns nothing. In this section, we present such a definition, and we prove that our protocol fulfills this definition of privacy even in the presence of a *malicious corrupted party*. We believe that this formalization is of independent interest, and could be used to make similar claims regarding other information-theoretic protocols like [2] and [4, 5]; namely, that although they are only secure in the presence of semi-honest adversaries, they are in fact *private in the presence of malicious adversaries*.

Before proceeding, we stress that a definition of privacy is strictly weaker than standard definitions of security for malicious adversaries. Most notably, *correctness* is not guaranteed and a malicious server may tamper with the output. In settings where the adversary may receive some feedback about the output, this may also reveal information about the input. Thus, our claim of privacy is only with respect to a malicious server who receives no information about the output.

Defining security. Let $\text{VIEW}_{\mathcal{A},I,\pi}(\vec{v},\kappa)$ denote the view of an adversary $\mathcal{A}$ who controls parties $\{P_i\}_{i\in I}$ (with $I \subset [n]$) in a real execution of the n-party protocol π, with inputs $\vec{v} = (v_1, \ldots, v_N)$ and security parameter κ. We stress that in this setting, the vector of inputs $\vec{v}$ is of length N and N may be much longer (or shorter) than the number of parties n running the protocol. This is because N refers to the number of inputs and so the number of clients, whereas n denotes the number of servers running the actual protocol. In addition, the servers do not receive for input any of the values in $\vec{v}$ but rather they each receive *secret shares* of the value. Formally, one should specify the secret sharing method. However, for generality, we do not define any specific secret sharing scheme and rather define that for every v_j in $\vec{v}$, random $v_j^1, \ldots, v_j^n$ are chosen under the constraint that $\sum_{\ell=1}^n v_j^\ell = v_j$, and each server P_j is given the share v_j^ℓ (for every $1 \le j \le N$).

Loosely speaking, a protocol is private in the presence of one malicious corrupted party if the view of the corrupted party when the input is $\vec{v}$ is computationally indistinguishable from its view when the input is $\vec{v}'$. In order to rule out a trivial protocol where nothing is exchanged, we also require correctness, which means that when all parties are honest they obtain the correct output.

DEFINITION 4.1. *Let $f : (\{0,1\}^*)^N \to (\{0,1\}^*)^N$ be an N-party functionality and let π be an n-party protocol. We say that π t-privately computes f in the client-server model in the presence of malicious adversaries if it is* correct *and if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$, every $I \subset [n]$ with $|I| \le t$, and every two series of length-N vectors $V_1 = \{\vec{v}_\kappa^1\}, V_2 = \{\vec{v}_\kappa^2\}$*

$$\left\{\text{VIEW}_{\mathcal{A},I,\pi}(\vec{v}_\kappa^1,\kappa)\right\}_{\kappa\in\mathbb{N}} \stackrel{\text{c}}{\equiv} \left\{\text{VIEW}_{\mathcal{A},I,\pi}(\vec{v}_\kappa^2,\kappa)\right\}_{\kappa\in\mathbb{N}}$$

where for every $\kappa \in \mathbb{N}$, $\vec{v}_\kappa^1, \vec{v}_\kappa^2 \in (\{0,1\}^*)^N$ and all elements of $\vec{v}_\kappa^1$ and $\vec{v}_\kappa^2$ are of the same length.

We now prove that Protocol 2.3 fulfills Definition 4.1, when making the appropriate changes to the input (con-

verting vectors of length N into 3-way additive shares for the parties running Protocol 2.3).

THEOREM 4.2. *Let $f : ((\mathbb{Z}_{2^n})^*) \to ((\mathbb{Z}_{2^n})^*)$ be an N-party functionality and define the 3-party functionality g_f to be the function that receives 3 length-N input vectors that constitute additive-shares of the input vector $\vec{v}$ to f and outputs 3 length-N vectors that constitute additive-shares of $f(\vec{v})$. If F is a pseudorandom function, then Protocol 2.3 applied to function g_f 1-privately computes f in the client-server model in the presence of malicious adversaries.*

PROOF SKETCH: Correctness is also required for the semi-honest setting and this is therefore already implied by Theorem 3.4. In order to prove privacy, we need to show that the view of a malicious $\mathcal{A}$ controlling one party when the input is $\vec{v}$ is indistinguishable from its view when the input is $\vec{v}'$. We first prove that the views are *identical* when information-theoretic correlated randomness is used (as described in the beginning of Section 2.2).

First, intuitively, the views are identical with information-theoretic correlated randomness since all the adversary sees in every rounds is a random share. In order to see that this holds even when $\mathcal{A}$ is *malicious*, observe that each share sent to the adversary is masked by a new value obtained from the correlated randomness. Thus, irrespective of what $\mathcal{A}$ sends in every round, the value that it receives is a *random element*. Thus, its view is actually independent of the values that it sends.

Second, consider the view when Protocol 3.8 is used for computing $\mathcal{F}_{\text{CR}}$. In the setup phase, $\mathcal{A}$ sends some value k_i and receives k_{i-1}. However, the security of the protocol is proven based on the pseudorandomness of the function keyed by k_{i+1} that $\mathcal{A}$ does not see. Importantly to this case of malicious adversaries, k_{i+1} is chosen independently of what $\mathcal{A}$ sends. Furthermore, the parties generate randomness from this point on using local computation only. Thus, the values generated by the honest parties are pseudorandom, irrespective of what $\mathcal{A}$ sent. More formally, consider a reduction where $F_{k_{i+1}}$ is replaced by a truly random function f. Then, P_{i-1} computes $\alpha_{i-1} = F_{k_{i-1}}(id) - f(id)$ and P_{i+1} computes $\alpha_{i+1} = f(id) - F_{k_i}(id)$. Since k_i and k_{i-1} are fixed and independent of f, it follows that $\alpha_{i-1}, \alpha_{i+1}$ are random under the constraint that $\alpha_{i-1} + \alpha_{i+1} = -(F_{k_i}(id) - F_{k_{i-1}}(id)) = -\alpha_i$, as required. As we have stated, this holds irrespective of what value k_i that $\mathcal{A}$ sent, and $\mathcal{A}$ cannot influence the $\alpha_{i-1}, \alpha_{i+1}$ values computed since they involve local computation by the honest parties alone. Thus, the view in this case is indistinguishable from the view when the parties use information-theoretic correlated randomness. ■

5. EXPERIMENTAL RESULTS

5.1 Implementation and Bit-Slicing

We implemented the protocol for Boolean circuits in C++ using standard optimizations known for multiparty computation. One specific optimization that we found to be of great importance was the use of Intel intrinsics for bit slicing operations; we describe this in more detail here. Since our protocol is extremely simple, running a single computation is very wasteful both with respect to CPU and network utilization. A significant portion of this waste is due to the fact that our protocol processes single bits only, whereas modern processors work on larger objects. We ran our protocol on 12800 operations in parallel by batching 128 operations together and running 100 of these in parallel. This batching works by bit-slicing: the ith bit of input in 128 different inputs are sliced into a single string of length 128 (for each i). Likewise, the batched output bits need to be de-sliced into 128 separate outputs. This is a type of "matrix transpose" – see Figure 1 – and turns out to be very expensive. Indeed, a straightforward implementation of this bit slicing and de-slicing turned out to greatly dominate the overall execution time. Hence, we implemented fast bit-slicing and bit-deslicing methods using Intel SIMD intrinsics in order to reduce this cost.

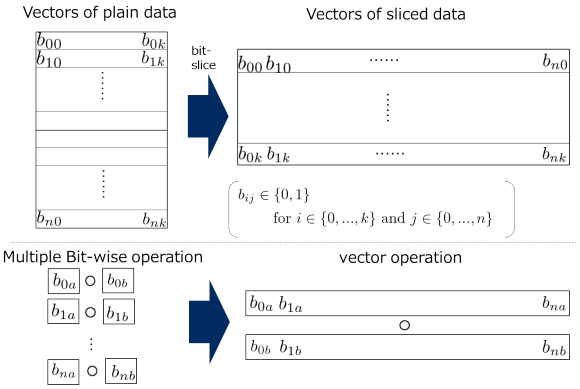

Figure 1: Bit-slice

The unit of our bit-slicing is 16 messages of length 8 bytes each (overall 128 bytes). Thus, we start with:

$$m_0 = (m_{0,0}, m_{0,1}, m_{0,2}, m_{0,3}, m_{0,4}, m_{0,5}, m_{0,6}, m_{0,7})$$
$$m_1 = (m_{1,0}, m_{1,1}, m_{1,2}, m_{1,3}, m_{1,4}, m_{1,5}, m_{1,6}, m_{1,7})$$
$$\dots$$
$$m_{15} = (m_{15,0}, m_{15,1}, m_{15,2}, m_{15,3}, m_{15,4}, m_{15,5}, m_{15,6}, m_{15,7}).$$

Then, we apply the Intel intrinsics "unpack" instruction 32 times to obtain 8 messages, each of length 16 bytes:

$$m'_0 = (m_{0,0}, m_{1,0}, \dots, m_{15,0})$$
$$m'_1 = (m_{0,1}, m_{1,0}, \dots, m_{15,1})$$
$$\dots$$
$$m'_7 = (m_{0,7}, m_{1,7}, \dots, m_{15,7}).$$

The unpack instruction treats the 128 bit register as 16 single-byte values (8 low and 8 high), and has instructions to interleave either the low or the high bytes. This process is actually **byte**-slicing (since the "transpose"-type operation is carried out at the byte level and not the bit level). See Figure 2 for a graphic description of this operation.

The next step is to further slice the messages to the bit level. We do this applying the Intel `movmskb` 64 times to obtain the bit-sliced inputs. This instruction creates a 16-bit mask from the most significant bits of 16 signed or unsigned 8-bit integers in a register and zeroes the upper bits. Thus, we are able to take the MSB of 16 bytes in a register in a *single cycle*, which is very fast. The `movmskb` instruction is depicted in Fig. 3.

We apply the `movmskb` operation to each m'_i from the first step (note that each m'_i consists of 16 bytes, exactly

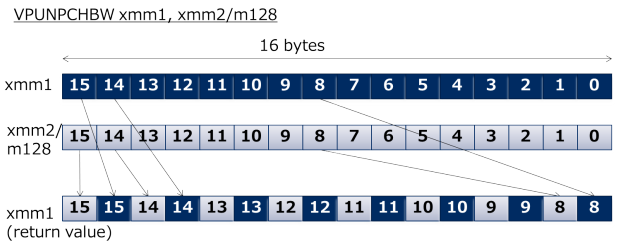

Figure 2: Unpack operation of AVX instruction set

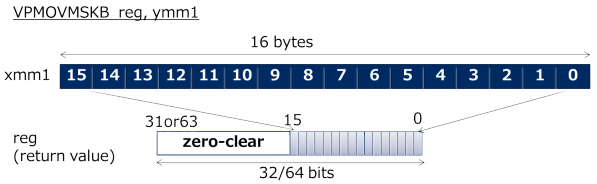

Figure 3: Moving masked bit operation of AVX instruction set

as needed for `movmskb`). These optimizations were crucial for obtaining the high performance reported in this paper.

5.2 Fast AES

We ran our implementation on a cluster of three mid-level servers connected by a 10Gbps LAN with a ping time of 0.13 ms. Each server has two Intel Xeon E5-2650 v3 2.3GHz CPUs with a total of 20 cores. We ran the implementation utilizing a different number of cores, from 1 through to 20. Each core was given 12800 computations which were carried out in parallel. (Since Intel intrinsics works on 128-bit registers, this means that inputs were sliced together in groups of 128 and then 100 of these were run in parallel by *each core*.) These computations can be with different keys since each MPC can have different inputs; this will be used in Section 5.3.

Observe that up to 10 cores, the throughput is stable at approximately 100,000 AES/sec per core. However, beyond 10 cores this begins to deteriorate. This is due to queuing between the kernel and the Network Interface Card (NIC). Specifically, when a single process utilizing a single CPU is used, that process has full control over the NIC. However, when multiple processes are run, utilizing high bandwidth, requests from each process are handled in a queue between the kernel and the NIC. This queuing increases network latency, and as each process spends more time waiting for communication, CPU usage drops by a noticeable percentage. It is possible to overcome this by bypassing the kernel layer and communicating directly with the NIC. One approach for achieving this appeared in [20].

We ran each experiment 5 times; this was sufficient due to the very low variance as can be seen in Table 2. The results represent a 95% confidence interval.

Cores	AES/sec	Latency	CPU %	Network
1	100,103 ± 1632	128.5 ± 2.1	73.3%	0.572
5	530,408 ± 7219	121.2 ± 1.7	62.2%	2.99
10	975,237 ± 3049	131.9 ± 0.4	54.0%	5.47
16	1,242,310 ± 4154	165.7 ± 0.4	49.5%	6.95
20	1,324,117 ± 3721	194.2 ± 0.9	49.6%	7.38

Table 2: Experiment results running AES-CTR. The CPU column shows the average CPU utilization per core, and the network column is in Gbps per server. Latency is given in milliseconds.

Recall that each core processed 12800 AES computations in parallel, and observe that with a latency of 129ms approximately 7 calls can be processed per second by each core. Thus, the approximate 100,000 AES computations per core per second are achieved in this way.

See Figures 4 and 5 for graphs showing the behavior of the implementation as higher throughputs are achieved.

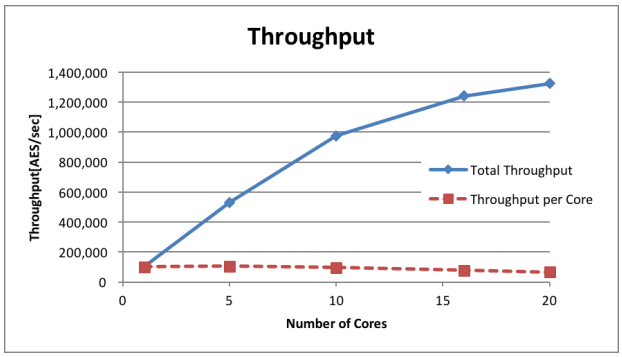

Figure 4: Throughput per core (AES computations)

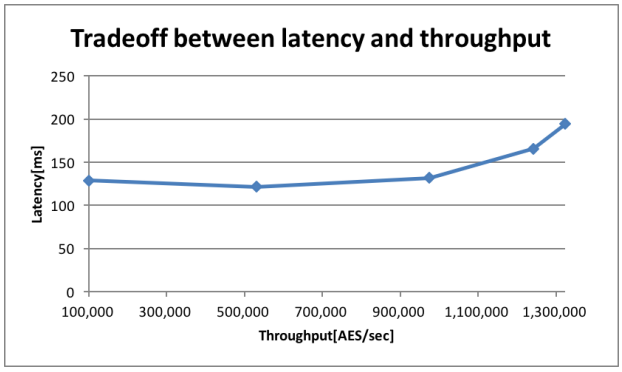

Figure 5: Latency versus throughput (AES)

Microbenchmarking. We measured the time spent on each part of the protocol, with the following results.

Protocol part	Percentage
Server bitslice and deslice	8.70%
AND and XOR gate computation	49.82%
Randomness generation	9.54%
Comm. delays between MPC servers	27.87%
Communication delays with king server	4.07%

We remark that the long communication delays are due to the fact that the communication topology of our implementation is a ring. Thus, each party waits for two other messages to be processed before it receives its next message. In order to reduce this waste, the randomness generation is run during this delay. Thus, if the randomness generation was "free", the communication delay would increase to 37.41% and it would not be any faster. This demonstrates that the efficiency improvements could be achieved by communicating in every step.

5.3 Kerberos KDC with Shared Passwords

In order to demonstrate the potential of our protocol, we incorporate it into a real application. Kerberos is used

for user authentication in many systems, most notably it is used by all Windows systems since Windows 2000. Kerberos uses the hashed user password as a key to encrypt a Ticket-Granting-Ticket (TGT) which contains a high-entropy cryptographic key which is used for all communications after the user logs in. In Kerberos, a server breach is particularly devastating since the hashed password is all that is needed for impersonating a user. This is because the TGT is encrypted with the hashed user password and sent to the user. Thus, an attacker knowing the hashed password alone can decrypt the TGT. Microsoft's Active Directory has suffered breaches in the past, and such a breach enables an attacker to impersonate every user in the organization.

In order to mitigate this risk, we consider a system where the hashed user passwords are XOR-shared between two servers (with different administrators), and secure multiparty computation is used to carry out the login authentication without ever reconstructing the hashed password. This makes it harder for an attacker to steal hashed passwords (needing to breach both servers) and also mitigates insider threats since no single administrator has access to the hashed user passwords. Since the ticket-granting-server's long-term key is also very sensitive, this is also protected in the same way. The architecture of the Kerberos solution is depicted in Figure 6.

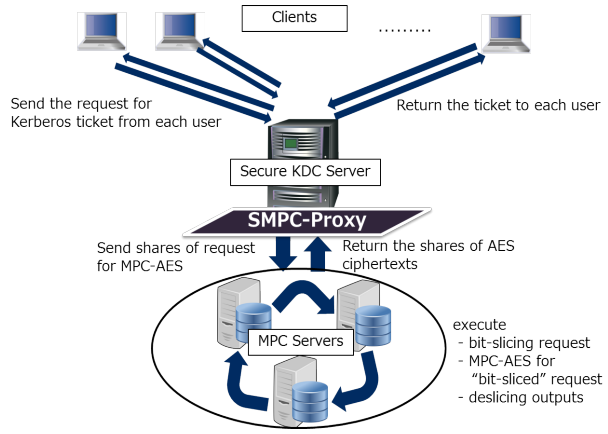

Figure 6: The Kerberos authentication using MPC

We took the Open Source MIT Kerberos and modified the encryption mode used to encrypt the TGT to counter mode. This is important since CBC mode does not enable parallel encryption and this would slow the encryption down significantly. In more detail, the authentication process in Kerberos has the following steps:

1. *Pre-authentication:* We use the PA-ENC-TIMESTAMP method, which means that the user encrypts the date using his hashed password as the key. This is a single AES block (and so ECB is used).

2. *TGT encryption:* A session key to be used by the user and ticket-granting server (TGS) to communicate later is generated. Then, the TGT (containing the client information and the session key) is generated and encrypted under the long-term key of the TGS. The TGT is 15 blocks of AES.

3. *Session-key and TGT encryption:* The session key and TGS are AES-encrypted with the user's hashed password.

Overall, the number of encryption blocks for a single user authentication is 33: one block for pre-authentication, 15 blocks for TGT encryption under the long-term key of the TGS, and 17 blocks for session-key and TGT encryption under the user key (this last encryption is 17 blocks due to the addition of the session key and header information).

In all of the above encryptions, when using the Kerberos encryption type `aes128-cts-hmac-sha1-96`, all of the encryption above is without HMAC authentication. (HMAC is only used for communication following these initial steps.) As we have mentioned, we implemented a Kerberos extension that uses counter mode instead of CBC (`cts` is CBC mode with ciphertext stealing). This is important for two reasons. First, CBC encryption cannot be parallelized and so each block must be encrypted after the previous block has been encrypted. In addition, the TGT cannot be encrypted under the user key until it has been encrypted under the long-term key of the TGS. However, when using counter mode, all of the AES computations can be carried out *in parallel*. Specifically, upon receiving a user authentication request together with a pre-authentication ciphertext, the following is carried out:

1. The servers running the secure computation protocol load the shares of the long-term key of the TGS and the shares of the user's key i.e., hash of the user's password).

2. Two random counters ctr_1 and ctr_2 are chosen.

3. 33 AES computations are run in parallel: a single AES decryption of the pre-authentication ciphertext, 15 AES encryptions of $ctr_1+1,\ldots,ctr_1+15$, and 17 AES encryptions of $ctr_2+1,\ldots,ctr_2+16$.

4. The preauthentication value is verified; if it is valid, then the server proceeds to the next step.

5. The output of the 15 AES encryptions using ctr_1 is XORed with the TGT.

6. The encrypted TGT from the previous step is concatenated with the session key and some header information. This is treated as a plaintext and XORed with the result of the 17 AES encryptions using ctr_2.

7. The result of the previous step along with ctr_1 and ctr_2 is sent to the user.

This flow enables all of the AES computations to be carried out in parallel, yielding a latency of approximately 120 milliseconds. We remark that in order for the server to be able to process requests in bulk, a new set of AES encryptions is begun every 100 milliseconds. Thus, authentication requests are queued for at most 100 milliseconds (and on average 50ms) and then processed. This ensures that the overall latency (of a client) of processing an authentication request is approximately 200 milliseconds. This is a very reasonable time for an application like Kerberos where a user is involved in the authentication process.

Experimental results. In order to test our implementation, we ran the complete Kerberos login using the aforementioned cluster of three servers computing AES. The number of logins per second with a single core was 2,970, with 10 cores was 28,723 and with 16 cores was 36,521. Thus, our Kerberos implementation (that incorporates the extension

described above in MIT-Kerberos) is able to support a significant login storm of over 35,000 user logins per second. This is sufficient even for very large organizations (if more is needed, then this can be achieved by simply using two clusters instead of one). Beyond the number of logins per second, it is important to ensure that the latency is low; otherwise, users will have to wait too long at login. This is the reason that we designed the TGT-generation process in a way that enables full parallelism of the AES operations. Our results give an average latency of the AES encryption via MPC at 110ms, and an average latency at the client (over a LAN) of 232ms. The increased time in the client is due to additional work carried out both by the client and the KDC, and due to the fact that requests are processed every 100ms.

Acknowledgements

We express our thank to Assi Barak and Felipe Zimmerle for their crucial help and contribution to the implementation and experimental results.

6. REFERENCES

[1] G. Asharov and Y. Lindell. A Full Proof of the BGW Protocol for Perfectly-Secure Multiparty Computation. To appear in *J. of Cryptology*.

[2] M. Ben-Or, S. Goldwasser, A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *STOC 1988*: 1-10

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

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

[5] D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec. 11(6):* 403-418, 2012.

[6] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. In the *Journal of Cryptology*, 13(1):143-202, 2000.

[7] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145, 2001.

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

[9] R. Cramer, I. Damgård and Y. Ishai. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In the 2*nd TCC*, Springer (LNCS 3378), pages 342–362, 2005.

[10] I. Damgård and M. Keller. Secure multiparty AES. In *Financial Cryptography*, Springer (LNCS 6052), pages 367–374, 2010.

[11] O. Goldreich: *Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press 2004

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

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

[14] Y. Ishai and E. Kushilevitz. On the Hardness of Information-Theoretic Multiparty Computation. In *EUROCRYPT 2004*, Springer (LNCS 3027), pages 439–455, 2004.

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

[16] L. Kerik, P. Laud and J. Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *4th WAHC*, 2016.

[17] E. Kushilevitz, Y. Lindell and T. Rabin. Information-Theoretically Secure Protocols and Security Under Composition. In the *SIAM Journal on Computing*, 39(5): 2090-2112, 2010.

[18] J. Launchbury, I.S. Diatchki, T. DuBuisson and A. Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In *ACM ICFP'12*, pages 189–200, 2012.

[19] S. Laur, R. Talviste and J. Willemson. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *ACNS'13*, Springer (LNCS 7954), pages 84–101, 2013.

[20] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah and H Fugal. Fastpass: a centralized "zero-queue" datacenter network. In *SIGCOMM 2014*, pages 307–318, 2014

[21] T. Rabin, M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract). *STOC 1989*: 73-85

[22] J. Randmets. Personal comm. – AES performance on the new Sharemind cluster. May, 2016.

[23] R. Talviste. *Applying Secure Multi-Party Computation in Practice.* Ph.D dissertation, Univ. of Tartu, 2016.

[24] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[25] A. Yao. How to Generate and Exchange Secrets. In the 27*th 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. *EUROCRYPT*, pages 220–250, 2015.

[27] Sharemind, Cybernetica. `https://sharemind.cyber.ee`.