A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority[*]

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

Ariel Nof
Bar-Ilan University
nofarie@cs.biu.ac.il

ABSTRACT

Protocols for secure multiparty computation enable a set of parties to compute a function of their inputs without revealing anything but the output. The security properties of the protocol must be preserved in the presence of adversarial behavior. The two classic adversary models considered are *semi-honest* (where the adversary follows the protocol specification but tries to learn more than allowed by examining the protocol transcript) and *malicious* (where the adversary may follow any arbitrary attack strategy). Protocols for semi-honest adversaries are often far more efficient, but in many cases the security guarantees are not strong enough.

In this paper, we present a new efficient method for "compiling" a large class of protocols that are secure in the presence of semi-honest adversaries into protocols that are secure in the presence of malicious adversaries. Our method assumes an honest majority (i.e., that $t < n/2$ where t is the number of corrupted parties and n is the number of parties overall), and is applicable to many semi-honest protocols based on secret-sharing. In order to achieve high efficiency, our protocol is *secure with abort* and does not achieve fairness, meaning that the adversary may receive output while the honest parties do not.

We present a number of instantiations of our compiler, and obtain protocol variants that are very efficient for both a small and large number of parties. We implemented our protocol variants and ran extensive experiments to compare them with each other. Our results show that secure computation with an honest majority can be practical, even with security in the presence of malicious adversaries. For example, we securely compute a large arithmetic circuit of depth 20 with 1,000,000 multiplication gates, in approximately 0.5 seconds with three parties, and approximately 29 seconds with 50 parties, and just under 1 minute with 90 parties.

1 INTRODUCTION

1.1 Background

Protocols for secure computation enable a set of parties with private inputs to compute a joint function of their inputs while revealing nothing but the output. The security properties typically required from secure computation protocols include *privacy* (meaning that nothing but the output is revealed), *correctness* (meaning that the output is correctly computed), *independence of inputs* (meaning that a party cannot choose its input as a function of the other parties' inputs), *fairness* (meaning that if one party gets output then so do all), and *guaranteed output delivery* (meaning that all parties always receive output). Formally, the security of a protocol is proven by showing that it behaves like an ideal execution with an incorruptible trusted party who computes the function for the parties [10]. In some cases, fairness and guaranteed output delivery are not required. This is standard in the case of no honest majority (since not all functions can be computed fairly without an honest majority), but can also be the case otherwise in order to aid the construction of highly efficient protocols.

Protocols for secure computation must remain secure in the face of adversarial behavior. There are many parameters determining the adversary:

- **Adversarial behavior:** If the adversary is *semi-honest*, then it follows the protocol specification but may try to learn more than is allowed by inspecting the protocol transcript. If the adversary is *malicious*, then it may follow an arbitrary attack strategy in its attempt to break the protocol.
- **Adversarial power:** If the protocol is guaranteed to remain secure even if the adversary is computationally unlimited, then the protocol is said to be *information-theoretically secure*. If the adversary is bounded to probabilistic polynomial-time, then the protocol is *computationally secure*.
- **Number of corruptions:** Denote by t the number of corrupted parties and by n the overall number of parties. There are typically three main thresholds that are considered in the literature: $t < n$ (meaning any number of parties may be corrupted), $t < n/2$ (meaning that there is an honest majority), and $t < n/3$ (meaning that less than a third of the parties are corrupted).
- **Corruption strategy:** If the set of corrupted parties is determined at the onset of the protocol, then we say that the adversary is *static*. If the adversary can determine who to corrupt throughout the execution, then it is *adaptive*.

In the late 1980s, it was shown that any function can be securely computed. This was demonstrated in the computational setting for any $t < n$ [24, 36], in the information-theoretic setting with $t < n/3$ [7, 11], and in the information-theoretic setting with $t < n/2$ assuming a broadcast channel [34]. These feasibility results demonstrate that secure computation is possible. However, significant improvements are necessary to make it efficient enough to use in practice.

[*] 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 '17, Dallas, TX, USA
© 2017 ACM. 978-1-4503-4946-8/17/10…$15.00
DOI: 10.1145/3133956.3133999

1.2 Our Protocol Framework

In this paper, we consider the problem of constructing highly efficient protocols that are secure in the presence of *static malicious adversaries* who control at most $t < n/2$ corrupted parties. Our protocol is fundamentally information-theoretic, but some efficient instantiations are computational. In the aim of achieving high efficiency, our protocols do not achieve fairness (even though this is possible in our setting where $t < n/2$).

We construct two protocol frameworks that can be used to compile a large class of semi-honest protocols based on secret sharing (for the case of $t < n/2$) into protocols that are secure in the presence of malicious adversaries. Our protocols are designed for *arithmetic circuits* over a field $\mathbb{F}$. Our protocol compiler works by computing a circuit using a semi-honest protocol and then running a verification step that ensures that the honest parties detect cheating with high probability. We follow the Beaver multiplication triple methodology [4] in which the parties generate shares of triples (a, b, c) where $c = a \cdot b$. Such triples are very useful as they can be used to carry out multiplication efficiently, and to verify that multiplications were carried out correctly. We present two ways of verifying multiplication, and show how to construct a tightly optimized protocol that uses these methods. Our verification methods involve inserting fresh randomness into the process, which enables us to detect cheating *even* if the multiplication triples are incorrect. Previous works that used the Beaver method required the multiplications triples to be correct in order to be able to detect cheating. The fact that we are able to detect cheating even if they are *not correct* enables much faster generation of these triples. We believe that this technique is of independent interest and will be useful in other protocols as well.

We present two frameworks; one that is better suited when working over "large" fields and one that is better suited to "small" fields. Then, we provide efficient instantiations of the semi-honest protocols, that when plugged into our frameworks yield highly efficient protocols that are secure for malicious adversaries. We consider two main instantiations: one based on Shamir sharing for any number of parties, and one based on replicated secret sharing for the specific case of three parties. For the case of Shamir sharing, we have several different instantiations; for example, multiplication can be computed using [25] or [17], verification can be carried out in one of two methods, and so on. We focus on two of the instantiations here: the first that is best for a small number of parties, and the second that is best for a large number of parties. We summarize our best instantiations with the following theorems. Our protocols have statistical error $2^{-\sigma}$. For the sake of simplicity, we consider the case that the field for the arithmetic circuit has the property that $|\mathbb{F}| > 2^{\sigma}$ (this is not necessary for our framework, but it yields the highest efficiency).

The first theorem is suited for a large number of parties, and is based on Shamir sharing.

Theorem 1.1 (large number of parties). *Let n be any number of parties, and let f be an n-party functionality. Then, there exists a protocol π that computes f with computational security in the presence of a malicious adversary controlling up to $t < n/2$ corrupted parties, where each party sends* 42 *field elements per multiplication gate.*

The next theorem is also based on Shamir sharing. However, it uses the PRSS methodology of [13] for generating random shares that is only computationally secure. In addition, it involves local work by each party that is exponential in the number of parties. Thus, it is only efficient for small values of n.

Theorem 1.2 (small number of parties). *Let n be any number of parties, and let f be an n-party functionality. Then, there exists a protocol π that computes f with computational security in the presence of a malicious adversary controlling up to $t < n/2$ corrupted parties, where each party sends* $5(n-1)$ *field elements per multiplication gate.*

Note that the protocol of Theorem 1.1 is preferable for a large number of parties since the number of elements sent per gate is constant (this holds even if PRSS is not used in the protocol for a small number of parties). Observe that when $n > 9$ it holds that $5(n-1) > 42$, and thus the constant is smaller than the linear function. Finally, we state our result that is specific to 3 parties and uses replicated sharing.

Theorem 1.3 (three parties). *Let f be a* 3*-party functionality. Then, there exists a protocol π that computes f with computational security in the presence of a malicious adversary controlling up to* 1 *corrupted party, where each party sends* 4 *field elements per multiplication gate.*

Observe that for the case of 3 parties, in the protocol of Theorem 1.2 each party sends 10 elements per multiplication gate, versus just 4 elements in the protocol of Theorem 1.3.

1.3 Experimental Results

We implemented our protocol versions in C++ and ran our protocols on Azure in a single region, with a ping time of approximately 1ms. Each machine is a 2.4GHz Intel Xeon E5-2673 v3, with 4 cores and 8GB RAM; our code is single-threaded so does not utilize more than one core. We ran extensive experiments to analyze the efficiency of the different protocols for different numbers of parties. All of our protocols scale linearly in the size and depth of the circuit, and we therefore ran all of our experiments on a depth-20 arithmetic circuit over a 61-bit field with 1,000,000 multiplication gates. Our experiments show that our protocols have very good performance for all ranges of numbers of parties. In particular, this large circuit can be computed in half a second with three parties, 4 seconds with 9 parties, 29 seconds with 50 parties and 59 seconds with 90 parties. Thus, our protocols can be used in practice to compute arithmetic computations (like joint statistics) between many parties, while providing malicious security.

1.4 Related Work

There has been a huge amount of work that focuses on improving the efficiency of secure computation protocols. This work is roughly divided up into constructions of *concretely efficient* and *asymptotically efficient* protocols. Concretely efficient protocols are often implemented and aim to obtain the best overall running time, even if the protocol is not asymptotically optimal (e.g., it may have quadratic complexity and not linear complexity, but for a small number of parties the constants are such that the quadratic protocol is faster). Asymptotically efficient protocols aim to reduce the cost of certain parts of the protocols (rounds, communication complexity, etc.), and are often not concretely very efficient. However, in many cases, asymptotically efficient protocols provide techniques that inform the construction of concretely efficient protocols.

As explained earlier, there are two main thresholds considered for an honest majority: $t < n/2$ and $t < n/3$. For the case of $t < n/3$, it was shown in [17] and [6] how to achieve unconditional and perfect security in the presence of a malicious adversary, with communication that grows linearly with the number of parties. The VIFF protocol [14], provided the first concretely efficient implementation for $t < n/3$. VIFF has quadratic communication complexity, and thus is suitable for a small number of parties. Although their execution times are 3-4 orders of magnitude slower than ours (for 4 players, they report on 200 multiplications per second), we estimate that an optimized implementation on modern machines and a network identical to ours would achieve similar times.

We stress, however, that for malicious adversaries, it is considerably harder to achieve a similar level of efficiency with $t < n/2$ since there is far less redundancy in the secret sharing. (To be exact, with $t < n/3$ it is possible to multiply two polynomials together and the degree is still less that $2n/3$, in contrast to the case of $t < n/2$.) Thus, techniques like those used in [6] do not apply to our case. Clearly, it is always best to guarantee security against more powerful adversaries, and thus a protocol that is secure for $t < n/2$ is preferable to a protocol that requires $t < n/3$. Furthermore, in practice, this is of great significance for a small number of parties. Specifically, for $n = 3$ it is not possible to run a protocol that requires $t < n/3$, and for $n = 9$ a protocol with $t < n/2$ can tolerate up to 4 corrupted parties whereas a protocol with $t < n/3$ can tolerate only 2 corrupted parties. Thus, protocols that are secure for $t < n/2$ are preferable.

For $t < n/2$ corrupted parties, it is much harder to achieve linear complexity without relying on cryptographic assumptions. This was first achieved in [23] using additively-homomorphic encryption [33] which is far from concretely efficient. Secure computation that scales linearly with the number of parties was also presented in [15], but this protocol requires that $t < n/2 - O(n)$, and its focus is on asymptotic and not concrete efficiency. Thus, both protocols, although achieving full security, are much more expensive than ours. The protocol of [5] is the only protocol, to the best of our knowledge, that achieves near-linear communication complexity per multiplication gate, namely $O(n \log n)$, in the information-theoretic model and with full security. This protocol uses expensive techniques for player elimination and computing authentication tags, and is therefore also not concretely efficient. However, it achieves full security, whereas we do not guarantee fairness. Recently, [21, 22] show how to compile semi-honest protocols into maliciously secure protocols for a variety of parameters, and also achieve linear complexity for the case of $t < n/2$. No concrete cost analysis or implementation was given in [21, 22], and thus it is difficult to estimate the concrete efficiency on specific instantiations. However, the constants are significantly higher than ours. At the core, [22] requires 16 semi-honest multiplications per gate in contrast to 6 in our best protocol; in addition, [22] requires more communication. We stress, however, that [21, 22] is far more general and also works in the dishonest majority setting, unlike our compiler. The work of [12] also considers the setting of $t < n/2$, malicious adversaries and arithmetic circuits. Their protocol works in a very different way to ours. In particular, they do not follow the Beaver triples methodology [4] in order to achieve security for malicious adversaries. Rather, they carry out the computation redundantly on the input and on the input multiplied by a secret (shared) random value. Correctness of the computation is validated by verifying that the different computation outputs fulfill a given equation. In [12], the majority of the work is in the circuit computation phase and the verification step is cheap, whereas in our work the majority of the work is in the verification and the circuit computation phase is cheap. The setting of $t < n/2$ and malicious adversaries was also studied in [1, 31], including implementations. However, they consider only three parties and Boolean circuits.

Finally, we remark that concrete efficiency for the case of a dishonest majority has also been studied [8, 16, 18, 28, 32]. This setting is considerably harder, and thus protocols are naturally far less efficient. The state-of-the-art MASCOT protocol [28] achieves a rate of below 1,000 multiplication gates (for 3 parties), which is orders of magnitude slower than what can be achieved in the honest-majority setting.

2 PRELIMINARIES

Notation. Let $P_1, ..., P_n$ denote the n parties participating in the computation, and let t denote the number of corrupted parties. Since we assume an honest majority, it follows that $t < \frac{n}{2}$. Finally, let $\mathbb{F}$ be a finite field and let $|\mathbb{F}|$ denote its size.

2.1 Threshold Secret Sharing

A secret sharing scheme with threshold t enables n parties to share a secret $v \in \mathbb{F}$ among n parties so that no subset of t parties can learn any information about it, whereas any subset of $t + 1$ parties can reconstruct it. We require that the scheme used in our protocol supports the following procedures:

- share(v): In this procedure, a dealer shares a value $v \in \mathbb{F}$. For simplicity, we consider *non-interactive* secret sharing, and thus there exists a probabilistic algorithm D that receives v (and some randomness) and outputs shares $v_1, \ldots, v_n$, where v_i is the share intended for party P_i. We denote the sharing of a value v by $[v]$. We stress that if the dealer is corrupted, then the shares received by the parties may not be correct. Nevertheless, we abuse notation and say that the parties hold shares $[v]$ even if these are not correct. Informally, we call a sharing correct if it defines a single valid secret; we will define this more formally below.
- reconstruct($[v], i$): Given a sharing of v and an index i held by the parties, this interactive protocol guarantees that if $[v]$ is not correct (see formal definition below), then P_i will output $\perp$ and abort. Otherwise, if $[v]$ is correct, then P_i will either output v or will abort.
- open($[v]$): Given a sharing of v held by the parties, this procedure guarantees that at the end of the execution, if $[v]$ is not correct, then *all* the honest parties will abort. Otherwise, if $[v]$ is correct, then each party will either output v or will abort. Clearly, open can be run by any subset of $t + 1$ or more parties. We require that if any subset J of $t + 1$ honest parties output a value v, then any superset of J will output either v or $\perp$ (but no other value).

- local operations: Given correct sharings $[u]$ and $[v]$ and a scalar $\alpha \in \mathbb{F}$, the parties can generate correct sharings of $[u+v]$, $[\alpha \cdot v]$ and $[v+\alpha]$ using local operations only (i.e., without any interaction). We denote these local operations by $[u]+[v]$, $\alpha \cdot [v]$, and $[v]+\alpha$, respectively.

We now define what it means for a sharing to be correct. The natural way to define this is to say that the honest parties' shares are the valid output of an execution of D. Formally, let $H \subseteq \{P_1,\ldots,P_n\}$ denote the set of honest parties. Then, a sharing $[v]=(v_1,\ldots,v_n)$ is correct if there exists a value v' and randomness r such that for every $i \in H$ it holds that v_i is the ith share output by the dealer algorithm D with input v' and randomness r. Although natural, we will actually require only a more relaxed version of correctness. For a subset of honest parties J of size $t+1$, we denote by $\mathsf{val}([v])_J$ the value obtained by these parties after running the open procedure (and where no corrupted parties or additional honest parties participate). Note that $\mathsf{val}([v])_J$ may equal $\perp$ if the shares held by the honest parties are not valid. We are now ready to formally define correctness.

Definition 2.1. Let $H \subseteq \{P_1,\ldots,P_n\}$ denote the set of honest parties. A sharing $[v]$ is correct if there exists a value $v' \in \mathbb{F}$ ($v' \neq \perp$) such that for *every* $J \subseteq H$ with $|J|=t+1$ it holds that $\mathsf{val}([v])_J = v'$.

Observe that if a sharing $[v]$ is incorrect then either **(a)** there exists a subset of $t+1$ honest parties $J \subseteq H$ such that $\mathsf{val}([v])_J=\perp$, or **(b)** there exist two subsets of $t+1$ honest parties $J_1,J_2 \subseteq H$ such that $\mathsf{val}([v])_{J_1} \neq \mathsf{val}([v])_{J_2}$. In case (a) occurs we say that $[v]$ is *invalid*; in case (b) occurs we say that $[v]$ is *value-inconsistent*. If a sharing is not invalid, then we say that it is *valid.*

In some cases, like Shamir's secret sharing, for every J of size $t+1$ the shares of the parties define a valid value. However, in some other cases, like replicated secret sharing, this is not necessarily the case. In such cases, we need an additional requirement on the secret sharing scheme, as follows. A secret sharing scheme is robustly-linear if for every pair of *invalid* shares $[u]$ and $[v]$ (as defined above) it holds that there exists a unique $\alpha \in \mathbb{F}$ such that $\alpha \cdot [u]+[v]$ is valid when computed via local operations (and thus when α is chosen randomly, $\alpha \cdot [u]+[v]$ is almost always invalid). Note that if a secret sharing scheme has no invalid sharings, like Shamir's secret sharing, then it is trivially robustly-linear.

We now prove some useful claims about such secret sharing schemes. The first claim is a triviality and follows from the fact that addition and scalar multiplication are local operations, and so the honest parties' values are not influenced by the adversary.

Claim 2.2. *If $[u]$ and $[v]$ are correct sharings, then for every $\alpha \in \mathbb{F}$ it holds that $[w]=\alpha \cdot [u]+[v]$ is correct.*

We derive the following corollary by taking the contrapositive:

Corollary 2.3. *Let $[u]$ and $[v]$ be sharings and let $\alpha \in \mathbb{F}$. If $[w]=\alpha \cdot [u]+[v]$ is not correct, then $[u]$ or $[v]$ are incorrect.*

The following lemma is used for checking correctness.

Lemma 2.4. *Let $[u]$ be an incorrect sharing of a robustly-linear secret sharing scheme and let $[v]$ be any sharing. Then, the probability that $[w]=\alpha \cdot [u]+[v]$ is a correct sharing where $\alpha \in_R \mathbb{F} \setminus \{0\}$ is randomly chosen, is at most $\frac{1}{|\mathbb{F}|-1}$.*

Proof. First, consider the case that $[v]$ is correct, and assume by contradiction that $[w]=\alpha \cdot [u]+[v]$ is correct. By Claim 2.2, this implies that $[w]-[v]$ is correct and thus that $[u]=\alpha^{-1} \cdot ([w]-[v])$ is correct, in contradiction.

Next, consider the case that $[v]$ is not correct, and thus both $[u]$ and $[v]$ are incorrect. There are two cases regarding the incorrectness of $[u]$ (recall that $[u]$ may be invalid or value-inconsistent, as defined above):

(1) *Case 1 – $[u]$ is invalid:* Let J be a subset of $t+1$ honest parties such that $\mathsf{val}([u])_J=\perp$. There are two subcases:
   (a) If $[v]$ is valid, then $\mathsf{val}([v])_J \neq \perp$, and by the same argument as Corollary 2.3 it must hold that $\mathsf{val}([w])_J=\perp$. Thus, $[w]$ is invalid and so incorrect.
   (b) If $[v]$ is invalid, then we have that both $[u]$ and $[v]$ are invalid. By the assumption that the secret sharing scheme is robustly-linear, there exists a unique α such that $[w]=\alpha \cdot [u]+[v]$ is valid. Since $\alpha \in_R \mathbb{F} \setminus \{0\}$, we have that the unique α making the result valid is chosen with probability only $\frac{1}{|\mathbb{F}|-1}$.

(2) *Case 2 – $[u]$ is value-inconsistent:* Let $u_1,u_2 \in \mathbb{F}$ be distinct values and let J_1 and J_2 be subsets of $t+1$ honest parties such that $\mathsf{val}([u])_{J_1}=u_1$ and $\mathsf{val}([u])_{J_2}=u_2$. If $\mathsf{val}([v])_{J_1}=\perp$, then by the same reasoning as in Corollary 2.3, it follows that $\mathsf{val}(\alpha \cdot u+v)_{J_1}=\perp$ and thus $\mathsf{val}([w])$ is invalid; likewise if $\mathsf{val}([v])_{J_2}=\perp$. We therefore proceed to analyze the case that for some $v_1,v_2 \in \mathbb{F}$ it holds that $v_1=\mathsf{val}([v])_{J_1}$ and $v_2=\mathsf{val}([v])_{J_2}$. There are two subcases:
   (a) If $v_1=v_2$ then $\alpha \cdot u_1+v_1 \neq \alpha \cdot u_2+v_2$ (this holds since $v_1=v_2$, $\alpha \neq 0$ and $u_1 \neq u_2$), and thus $\mathsf{val}(\alpha \cdot [u]+[v])_{J_1} \neq \mathsf{val}(\alpha \cdot [u]+[v])_{J_2}$, implying that $[w]$ is incorrect.
   (b) If $v_1 \neq v_2$ then $\mathsf{val}([w])_{J_1}=\mathsf{val}([w])_{J_2}$ if and only if $\alpha \cdot u_1+v_1=\alpha \cdot u_2+v_2$ which holds if and only if $\alpha=(v_2-v_1)\cdot(u_1-u_2)^{-1}$. (Note that this is well defined since $u_1 \neq u_2$.) Since α is random in $\mathbb{F} \setminus \{0\}$, this equality holds with probability only $\frac{1}{|\mathbb{F}|-1}$.

We therefore conclude that $[w]$ is correct with probability at most $\frac{1}{|\mathbb{F}|-1}$, as required. □

2.2 Definitions

Privacy for malicious adversaries. Our protocol works by running a multiplication protocol (for multiplying two shares $[x]$ and $[y]$) that is secure for semi-honest adversaries, and then compiling it into a protocol that is secure for malicious adversaries by adding a verification step that allows the honest parties to detect cheating. For our compiler, it is necessary that the semi-honest multiplication protocol used achieves privacy in the presence of malicious adversaries. We use the formulation of this notion as provided in [3]. As we will see, this condition is easily met by all standard secret sharing based semi-honest multiplication protocols.

Let $\textsc{View}_{\mathcal{A},I,\pi}(\vec{x},\kappa)$ denote the view of an adversary $\mathcal{A}$ who controls parties $\{P_i\}_{i \in I}$ (with $I \subset \{1,\ldots,n\}$) in a real execution of the n-party protocol π, with inputs $\vec{x}=(x_1,\ldots,x_n)$ and security parameter κ. Loosely speaking, a protocol is private in the presence of t malicious corrupted parties if the view of the corrupted parties when the input is $\vec{x}$ is computationally indistinguishable from its view when the input is $\vec{x}'$. 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 2.5. Let $f : \mathbb{F}^n \to \mathbb{F}^n$ be an n-party functionality and let π be an n-party protocol. We say that π t-privately computes f in the presence of malicious adversaries if it is *correct* and if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$, every $I \subset \{1, \ldots, n\}$ with $|I| \leq t$, and every two vectors $\vec{x}, \vec{x}' \in \mathbb{F}^n$

$$\left\{\textsc{View}_{\mathcal{A},I,\pi}(\vec{x},\kappa)\right\}_{\kappa\in\mathbb{N}} \overset{c}{\equiv} \left\{\textsc{View}_{\mathcal{A},I,\pi}(\vec{x}',\kappa)\right\}_{\kappa\in\mathbb{N}}$$

where all elements of $\vec{x}$ and $\vec{x}'$ are of the same length.

We say that a protocol is *private semi-honest* if it is private in the presence of malicious adversaries (as in Definition 2.5) and secure in the presence of semi-honest adversaries (under the standard definition of security).

Security in the presence of malicious adversaries. We use the standard ideal-real paradigm to prove security. We stress that our protocol provides security with abort only and not full security, despite the fact that an honest majority exists. Full definitions appear in the full version of this paper.

Security up to additive attacks. In one of the variants of our compiler (Section 3.4), we will need a stronger notion than just privacy. We say that a protocol is secure up to additive attacks in the presence of malicious adversaries if, in addition to privacy as defined above, it guarantees that a malicious adversary can only cheat by (obliviously) adding a value to the output.

Formally, consider the case that exactly $t + 1$ parties are honest. Then, define an ideal functionality for multiplication that receives shares of x, y from the honest parties and a value d from the adversary, and gives the parties shares of $x \cdot y + d$ for output. Then, a multiplication protocol is *secure up to additive attacks in the presence of a malicious adversary* if it securely computes this ideal functionality in the presence of malicious adversaries (under the standard definition). Note that we define it here for multiplication only, since that is what we need. More information about this model and definitions can be found in [21].

We stress that many semi-honest protocols based on secret sharing actually achieve security up to additive attacks. Specifically, it was proven in [21] that the BGW [7], GMW [24] and DN [17] semi-honest protocols, are all secure up to additive attacks. We utilize this in Section 6, where we present instantiations of our protocol.

3 SUB-PROTOCOLS AND BUILDING BLOCKS

In this section, we present the main building blocks that are used in our protocol.

3.1 Generating Random Value and Shares

Generating random shares. We use an ideal functionality $\mathcal{F}_{\text{rand}}$ that generates a sharing $[r]$ of a random value $r \in \mathbb{F}$ and provides each party with its share. We stress that in our protocol, the random shares that are generated need not to be correct, and thus it is not required that the protocol for generating random shares securely realizes $\mathcal{F}_{\text{rand}}$ as defined above. In fact, for our proofs to work, it suffices that the random shares are only valid, but not necessarily correct. However, for simplicity and clarity of the presentation, we use the stronger definition. As we will see in Section 6, in some instances this functionality can be securely realized at very low cost (even non-interactively). In one of our instantiations we will use the weaker variant; this will be discussed and justified in Section 6.1.

Generating random coins. The functionality $\mathcal{F}_{\text{rand}}$ can be used to securely compute a functionality $\mathcal{F}_{\text{coin}}$ that chooses a random element from $\mathbb{F} \setminus \{0\}$ and hands it to the parties. This functionality can be easily realized by having the parties call $\mathcal{F}_{\text{rand}}$ once and then open the result (if the sharing received from $\mathcal{F}_{\text{rand}}$ is to 0, then the parties simply repeat the process).

3.2 Correctness Check of Shares

In this section, we show how to verify that a series of m shares are correct. This is needed, for example, to ensure that the shares of the inputs provided by the parties at the onset of the protocol are correct. The verification method follows the approach of [12] and is formally described in Protocol 3.1. The idea is to choose random coefficients and then use them to compute a linear combination of the shares. Since the values of the coefficients are not known before the shares were generated, it follows that if there exists a sharing that is not correct, then the resulting share is correct with small probability (utilizing Lemma 2.4). In order to preserve the privacy of the shared values, the parties also add a random sharing to the linear combination, so that when the resulted sharing is open, nothing can be learned about the inputs of the parties.

PROTOCOL 3.1 (Batch Correctness Check of Shares).

- **Inputs:** The parties hold m shares $[x_1], \ldots, [x_m]$.
- **The protocol:**
  1. The parties call $\mathcal{F}_{\text{coin}}$ to receive random elements $\rho_1, \ldots, \rho_m \in \mathbb{F} \setminus \{0\}$.
  2. The parties call $\mathcal{F}_{\text{rand}}$ and obtain a sharing $[r]$.
  3. The parties locally compute
     $$[v] = \rho_1 \cdot [x_1] + \ldots + \rho_m \cdot [x_m] + [r]$$
  4. The parties run open$([v])$.
  5. If no abort message was received, then the parties output accept.

Lemma 3.2. *If there exists $j \in [m]$ such that $[x_j]$ is not correct, then the honest parties output* accept *in Protocol 3.1 with probability at most $\frac{1}{|\mathbb{F}|-1}$.*

Proof. Assume that there exists an index $j \in [m]$ such that $[x_j]$ is not correct. Let $[u] = \sum_{i\in[m]\setminus\{j\}} \rho_i \cdot [x_i] + [r]$, and observe that $[v] = \rho_j \cdot [x_j] + [u]$. Then, since $[x_j]$ is not correct, we have by Lemma 2.4 that $[v]$ is correct with probability at most $\frac{1}{|\mathbb{F}|-1}$ as required. Finally, recall that the open procedure guarantees that all parties abort (and so don't output accept) if an incorrect sharing is opened. □

We do not prove full security of this protocol; rather, we directly simulate it in the main protocol. Although this is a less modular approach, it actually significantly simplifies the proof. For example, formalizing this as an ideal functionality would require knowing how to generate the inconsistent messages for the case that some shares are inconsistent. This requires the simulator knowing the honest parties' shares in the case that they are inconsistent, which leads to an unnatural modeling of the functionality. In the full protocol simulation, the simulator already knows the honest parties' shares since it generated them itself (albeit as shares of 0, but the distribution over the messages in the simulation is not dependent on the value).

3.3 Triple Verification Based on the Open Procedure

A multiplication triple is a triple of shares $([a], [b], [c])$ with the property that $c = a \cdot b$. We say that a multiplication triple is *random* if $[a]$ and $[b]$ are sharings of random values in $\mathbb{F}$. We define correctness, as follows:

Definition 3.3. $([a], [b], [c])$ is a correct multiplication triple if $([a], [b]$ and $[c])$ are correct sharings and $c = a \cdot b$.

In this section, we show how to verify that a multiplication triple is correct *without revealing anything about its values*, by using (and wasting) an additional random multiplication triple. We use the same method as [16, 18], described in Protocol 3.4. The idea behind the protocol is as follows. Given shares of x, y, z and a, b, c the parties compute and open shares of $\rho = \alpha \cdot x + a$ and $\sigma = y + b$, where α is a random element generated at the beginning of the protocol. These values reveal nothing about x and y since a and b are random. In addition, from the way opening is defined, it follows that all the honest parties hold the same values for ρ and σ. As we will see in the proof below, if $([a], [b], [c])$ is a correct triple and $([x], [y], [z])$ is not, then $\alpha[z] - [c] + \sigma \cdot [a] + \rho \cdot [b] - \rho \cdot \sigma$ is either incorrect or a sharing of some element $d \neq 0$. Thus, by computing and opening this share, the honest parties can detect cheating and abort. In the case where $[a], [b], [c]$ is also incorrect, we show that the multiplication of $[z]$ by the random α ensures that the probability that the parties obtain a correct sharing of 0 is bounded by $\frac{1}{|\mathbb{F}|-1}$. As can be seen, the communication cost of this protocol is due to the three executions of the open procedure.

PROTOCOL 3.4 (Triple Verification Using Open).

- **Inputs:** The parties hold a triple $([x], [y], [z])$ to verify and an additional random triple $([a], [b], [c])$.
- **The protocol:**
  1. The parties call $\mathcal{F}_{\text{coin}}$ to generate a random $\alpha \in \mathbb{F} \setminus \{0\}$.
  2. Each party locally computes $[\rho] = \alpha \cdot [x] + [a]$ and $[\sigma] = [y] + [b]$.
  3. The parties run open$([\rho])$ and open$([\sigma])$, as defined in Section 2, to receive ρ and σ. If a party receives $\perp$ in an opening, then it sends $\perp$ to all the other parties and aborts.
  4. Each party locally computes
     $$[v] = \alpha[z] - [c] + \sigma \cdot [a] + \rho \cdot [b] - \rho \cdot \sigma.$$
  5. The parties run the open$([v])$ procedure to receive v. If a party receives $\perp$ in the opening, then it sends $\perp$ to all the other parties and aborts.
  6. Each party checks that $v = 0$. If not, then it sends $\perp$ to the other parties and aborts.
  7. If no abort messages are received, then the parties output accept.

We now prove the security guarantee provided by Protocol 3.4.

Lemma 3.5. *If $[x]$ and $[y]$ are correct shares, and $([x], [y], [z])$ is not a correct triple, then:*

(1) If $([a], [b], [c])$ is a correct triple, then all honest parties abort in Protocol 3.4 with probability 1.

(2) If $([a], [b], [c])$ is not a correct triple, then the honest parties output accept *in Protocol 3.4 with probability at most $\frac{1}{|\mathbb{F}|-1}$.*

Proof. Assume that $[x], [y]$ are correct shares and that the triple $([x], [y], [z])$ is not correct. First, observe that by the definition of the open procedure, if Step 3 concluded without the honest parties aborting, then it is guaranteed that $[\rho]$ and $[\sigma]$ are correct and that all the honest parties hold the same values ρ and σ. By Claim 2.2, this implies that $[a]$ and $[b]$ are correct shares as well. Thus, for the remainder of the proof, we can assume that $[x], [y], [a]$ and $[b]$ are all correct and that all the honest parties hold the correct values $\rho = \alpha \cdot x + a$ and $\sigma = y + b$. Recall that since $([x], [y], [z])$ is not correct, either $[z]$ is not correct or $z \neq x \cdot y$.

To prove Item (1) in the lemma, assume that $([a], [b], [c])$ is a correct multiplication triple, i.e., all the shares are correct and $c = a \cdot b$. Then, by Claim 2.2, $[u] = [c] + \sigma[a] + \rho[b] - \rho\sigma$ is correct. If $[z]$ is not correct, then $[v]$ is not correct (since $[z] = \alpha^{-1} \cdot ([v] + [u])$ and so by Claim 2.2, correctness of $[v]$ would imply correctness of $[z]$). Thus, the honest parties would all abort when running the open$([v])$ procedure. Else, if $[z]$ is correct but $z \neq x \cdot y$, then it holds that $z = x \cdot y + d$ for some $d \in \mathbb{F} \setminus \{0\}$. Then, when the parties locally compute $[v]$, it holds that:

$$\begin{aligned} [v] &= \alpha[z] - [c] + \sigma[a] + \rho[b] - \rho\sigma \\ &= [\alpha z - c + (y + b)a + (\alpha x + a)b - (\alpha x + a)(y + b)] \\ &= [\alpha z - c + ya + ab + \alpha xb + ab - \alpha xy - ay - \alpha xb - ab] \\ &= [\alpha z - \alpha xy + ab - c] \\ &= [\alpha(z - xy) - (c - ab)]. \end{aligned} \tag{1}$$

Since $c = a \cdot b$ and $z = x \cdot y + d$, it follows that $[v] = [\alpha \cdot d] \neq [0]$, and thus the honest parties abort in Step 6 with probability 1, as required.

Next, we prove Item (2) in the lemma, where $([a], [b], [c]$ is not a correct triple. Since we are guaranteed that $[a]$ and $[b]$ are correct, this means that either $[c]$ is not correct or that $c \neq a \cdot b$. Note that in both cases, we are guaranteed that $[u] = \sigma[a] + \rho[b] - \rho\sigma$ is correct. Now, if the parties do not abort in Step 5, then it follows that $[v] = \alpha \cdot [z] - [c] + [u]$ is a correct sharing of 0. Since $[u]$ is correct, then it follows from Claim 2.2 that $\alpha \cdot [z] - [c]$ is also correct. Using Claim 2.2 yet again, we have that this can happen only if both $[z]$ and $[c]$ are incorrect or both are correct. If they are both incorrect, then by Lemma 2.4, the probability that $\alpha \cdot [z] - [c]$ is correct is bounded by $\frac{1}{|\mathbb{F}|-1}$, as required. If both $[z]$ and $[c]$ are correct, we have that $c = a \cdot b + d_2$ and $z = x \cdot y + d_1$ for some $d_1, d_2 \in \mathbb{F} \setminus \{0\}$. Plugging this into Eq. (1), we have that $[v] = [\alpha \cdot d_1 - d_2]$. By Step 6 of the protocol, the honest parties output accept only if $\alpha \cdot d_1 - d_2 = 0$, which holds if only if $\alpha = d_2 \cdot d_1^{-1}$. Since α is distributed uniformly in $\mathbb{F} \setminus \{0\}$ and independent of d_1, d_2 (since it is chosen randomly *after* d_1 and d_2 are fixed), we have that the honest parties output accept with probability at most $\frac{1}{|\mathbb{F}|-1}$. We conclude that in all cases the probability that the protocol ends with the honest parties outputting accept is bounded by $\frac{1}{|\mathbb{F}|-1}$. □

As with Protocol 3.1, we do not prove the security of the protocol with respect to an ideal functionality. This is because such an ideal functionality would actually be very complex. For example, the ideal functionality would have to deal with the case that $[x]$, $[y]$ may not be correct. However, this case can never actually happen in our protocol (due to the way that $[x]$, $[y]$ are generated). Thus, it is preferable to not complicate matters with such a functionality.

3.4 Triple Verification Based on Multiplication Secure Up to Additive Attacks

In this section, we present a different protocol for verifying that a multiplication triple is correct. The protocol is similar to that of the previous section, except that here we use a multiplication protocol that is secure up to additive attacks (as defined in Section 2.2) instead of the open procedure. In more detail, in Protocol 3.4, the parties compute shares of

$$v = \alpha \cdot z - c + \sigma \cdot a + \rho \cdot b - \rho \cdot \sigma$$

where $\rho = \alpha x + a$ and $\sigma = y + b$, and verify that it equals 0. This is computed by first computing ρ, σ and opening them, and then using (local) scalar multiplication to obtain shares of v. However, openings are expensive operations, and in many cases are actually more expensive than semi-honest multiplication (this is true in the 3-party case, as well as when multiplying using the protocol of [17] as discussed in Section 6.1). Thus, in the protocol in this section, the parties compute shares of v by carrying out all of the multiplications using a secure protocol, and then only opening v. Thus, we construct two verification protocols that are based on different building blocks; one that is based on invocations of the open procedure and another that is based on a semi-honest multiplication protocol that is secure up to additive attacks [21]. As we will see later, this enables us to obtain different protocols that are better suited for different settings, depending on the number of parties, secret sharing method used, and so on.

We present the protocol based on multiplication in two steps. First, we introduce a protocol that still requires one opening for each triple verification; namely the parties open $[v]$ to verify that it equals 0. Next, we show how to check that many shares are all equal to 0 with a single opening. Since only one open procedure is run for many executions, this yields a protocol where the amortized cost per triple depends only on the cost of the multiplication and the random sharing generation protocols.

The first protocol is described in Protocol 3.6. As mentioned above, in this protocol, the values of ρ and σ are not opened as in the previous protocol. Instead, we compute the sharing

$$[\alpha] \cdot [z] - [c] + [\sigma] \cdot [a] + [\rho] \cdot [b] - [\rho] \cdot [\sigma]$$
$$= [\alpha] \cdot [z] - [c] + [\sigma] \cdot [a] - [\rho] \cdot [y]$$

using secure multiplication (where the equality holds because $\sigma = y + b$ and so $b - \sigma = -y$). Intuitively, this protocol is secure for the same reason as Protocol 3.4 that uses opening, since the multiplications are used to compute the same equation. However, we note that in this case we need to generate a random sharing $[\alpha]$ that is kept secret, and we cannot let its value be publicly known. To see why this is necessary, assume that $([a], [b], [c])$ is a correct multiplication triple and $[x]$, $[y]$ and $[z]$ are all correct sharings, but $z \neq x \cdot y$, i.e., $z = x \cdot y + d$ for some $d \neq 0$ known to the adversary. If the value of α is known to the adversary, then it knows that $\alpha \cdot [z]$ is a sharing of $\alpha \cdot xy + \alpha \cdot d$. Then, when computing $[u] = [c] + [\sigma] \cdot [a] + [\rho] \cdot [b] - [\rho] \cdot [\sigma]$ the adversary can cheat in one of the multiplications and make $[u]$ be a sharing of $(c + \sigma \cdot a + \rho \cdot b - \rho \cdot \sigma) + \alpha \cdot d$, thus causing $[v] = \alpha \cdot [z] - [u]$ to be a sharing of 0. In contrast, when α is not known, this attack cannot be carried out. See Protocol 3.6 for all details.

PROTOCOL 3.6 (TRIPLE VERIFICATION BASED ON MULTIPLICATION).

Let π_{mult} be a multiplication protocol that is secure up to additive attack, as described in Section 2.

- **Inputs:** The parties hold a triple $([x], [y], [z])$ to verify, and an additional random triple $([a], [b], [c])$.
- **The protocol:**
  1. The parties execute $\mathcal{F}_{\text{rand}}$ to generate a random sharing $[\alpha]$.
  2. The parties execute π_{mult} on $[x]$ and $[\alpha]$ to obtain $[\alpha \cdot x]$.
  3. Each party locally computes $[\rho] = [\alpha \cdot x] + [a]$ and $[\sigma] = [y] + [b]$.
  4. The parties execute π_{mult} on $[z]$ and $[\alpha]$ to obtain $[\alpha \cdot z]$.
  5. The parties execute π_{mult} on $[a]$ and $[\sigma]$ to obtain $[\sigma \cdot a]$.
  6. The parties execute π_{mult} on $[\rho]$ and $[y]$ to obtain $[\rho \cdot y]$.
  7. Each party locally computes
  $$[v] = [\alpha \cdot z] - [c] + [\sigma \cdot a] - [\rho \cdot y].$$
  8. The parties run the open$([v])$ procedure to receive v. If a party receives $\perp$ in the opening, then it sends $\perp$ to all the other parties and aborts.
  9. Each party checks that $v = 0$. if not, then it sends $\perp$ to the other parties and aborts.
  10. If no abort messages are received, then output accept.

Before proceeding, observe that if the parties are all honest then they accept, since

$$\begin{aligned} v &= \alpha \cdot z - c + \sigma \cdot a - \rho \cdot y \\ &= \alpha \cdot x \cdot y - a \cdot b + (y + b) \cdot a - (\alpha \cdot x + a) \cdot y \\ &= \alpha \cdot x \cdot y - a \cdot b + a \cdot y + a \cdot b - \alpha \cdot x \cdot y - a \cdot y \ = \ 0, \end{aligned}$$

as required. The protocol can be proven secure as long as all shares of the input are guaranteed to be *valid* (but not necessarily value-consistent or form correct triples). Thus, this method is only suited for protocols where validity is always guaranteed.

The following lemma shows the security provided by Protocol 3.6.

LEMMA 3.7. *Let $[a]$, $[b]$, $[c]$ be valid shares and let π_{mult} be a multiplication protocol that is secure up to additive attacks. If $[x]$,$[y]$ are correct shares and $[z]$ is valid, but $([x], [y], [z])$ is not a correct multiplication triple, then the honest parties output* accept *in Protocol 3.6 with probability at most negligibly greater than $\frac{1}{|\mathbb{F}|}$.*

PROOF. Assume that $[x]$,$[y]$ are correct shares, that $[z]$ is valid (meaning that all subsets of honest parties of size $t+1$ reconstruct to a value, and not to $\perp$) and that the triple $([x], [y], [z])$ is not correct (note that since x and y are well defined, then so is $x \cdot y$). This implies that either $[z]$ is value-inconsistent or that z is correct but $z \neq x \cdot y$. In both cases, there exists a subset J_0 of $t+1$ honest parties such that $\text{val}([z])_{J_0} = x \cdot y + \bar{d}$ where $\bar{d} \in \mathbb{F} \setminus \{0\}$. We consider the values $\text{val}([a])_{J_0} = a_{J_0}$ and $\text{val}([b])_{J_0} = b_{J_0}$ that the honest subset

J_0 would open for $[a]$ and $[b]$, respectively. Since $[c]$ is valid, it holds that $\mathsf{val}([c])_{J_0} = a_{J_0} \cdot b_{J_0} + d_1$ for some $d_1 \in \mathbb{F}$ (note that d_1 may equal 0).

In the protocol, the parties execute π_{mult} four times. Looking at the shares of parties in J_0, we obtain that there exist $d_2, \ldots, d_5 \in \mathbb{F}$ such that

$$\mathsf{val}([\alpha \cdot x])_{J_0} = \alpha \cdot x + d_2 \tag{2}$$

$$\mathsf{val}([\alpha \cdot z])_{J_0} = \alpha \cdot (x \cdot y + \bar{d}) + d_3 \tag{3}$$

$$\mathsf{val}([\sigma \cdot a])_{J_0} = (y + b_{J_0}) \cdot a_{J_0} + d_4 \tag{4}$$

$$\mathsf{val}([\rho \cdot y])_{J_0} = (\alpha \cdot x + d_2 + a_{J_0}) \cdot y + d_5. \tag{5}$$

The above holds by the assumption that π_{mult} is secure up to additive attacks and therefore these d_i values are well defined (and can be extracted by a simulator). By the properties of the open procedure, if all the honest parties output accept at the end of the protocol, then $[v]$ is a correct sharing of 0. In particular, this implies that for J_0 it holds:

$$\begin{aligned} \mathsf{val}([v])_{J_0} &= \mathsf{val}([\alpha \cdot z])_{J_0} - \mathsf{val}([c])_{J_0} + \mathsf{val}([\sigma \cdot a])_{J_0} - \mathsf{val}([\rho \cdot y])_{J_0} \\ &= \alpha \cdot (x \cdot y + \bar{d}) + d_3 - (a_{J_0} \cdot b_{J_0} + d_1) \\ &\quad + (y + b_{J_0}) \cdot a_{J_0} + d_4 - (\alpha \cdot x + d_2 + a_{J_0}) \cdot y - d_5 \\ &= \alpha \cdot \bar{d} - d_1 - d_2 \cdot y + d_3 + d_4 - d_5 \ = \ 0. \end{aligned}$$

This holds if and only if

$$\alpha \cdot \bar{d} = d_1 + d_2 \cdot y - d_3 - d_4 + d_5. \tag{6}$$

We claim that for x, y, z as in the lemma, Eq. (6) holds with probability at most negligibly greater than $\frac{1}{|\mathbb{F}|}$. To see this, assume by contradiction, that there exists an adversary $\mathcal{A}$ who participates in Protocol 3.6 and succeeds in causing Eq. (6) to hold with probability ϵ that is non-negligibly greater than $\frac{1}{\mathbb{F}}$. We will show by reduction that if such an $\mathcal{A}$ exists, then this contradicts the security of the secret sharing scheme. Let $[x], [y], [z], [a], [b], [c]$ be inputs for which $\mathcal{A}$ succeeds in having Eq. (6) holds with probability ϵ.

Before describing $\mathcal{S}$, let Π denote Protocol 3.6 where the π_{mult} executions are replaced with "ideal functionalities" computing multiplication under an additive attack; Π is a "hybrid" protocol containing both regular messages and ideal calls. As described in Section 2.2, this functionality receives d from the adversary, shares of two values x, y from the honest parties, and returns shares of $x \cdot y + d$ to all parties. Now, let $\mathcal{A}_\Pi$ be the adversary obtained from $\mathcal{A}$ by replacing the π_{mult} executions with these ideal calls; such an adversary exists by the sequential modular composition theorem of [9]. Accordingly, the probability that Eq. (6) holds when $\mathcal{A}_\Pi$ runs Protocol 3.6 on input $[x], [y], [z], [a], [b], [c]$ is negligibly close to ϵ.

We are now ready to describe $\mathcal{S}$ who breaks the security of the secret sharing scheme. Adversary $\mathcal{S}$ receives $t < n/2$ shares of a random element α, and internally plays the honest parties in set J_0 running the hybrid protocol Π with $\mathcal{A}_\Pi$, using inputs $[x], [y], [z], [a], [b], [c]$, up to and including Step 6. $\mathcal{S}$ obtains d_2, d_3, d_4, d_5 from $\mathcal{A}_\Pi$ during the internal simulation, computes $\bar{d} = z - x \cdot y$ and $d_1 = c - a \cdot b$. Then, $\mathcal{S}$ outputs

$$\alpha' = (d_1 + d_2 \cdot y - d_3 - d_4 + d_5) \cdot \bar{d}^{-1}.$$

By the assumption that $z \neq x \cdot y$, we have $\bar{d} \neq 0$ and thus $\bar{d}^{-1}$ is well defined. Furthermore, if Eq. (6) holds, then $\alpha' = \alpha$ and so $\mathcal{S}$ outputs the secret value, despite receiving less than $n/2$ shares. By the assumption on $\mathcal{A}$, we have that $\mathcal{S}$ succeeds with probability that is negligibly close to ϵ. This contradicts the security of the secret sharing scheme if ϵ is non-negligibly greater than $1/|\mathbb{F}|$. This completes the proof of the lemma. □

Removing the remaining call to the open *procedure.* Protocol 3.6 requires one execution of the open procedure. Stated differently, each verification of a triple requires one invocation of the open procedure. We now show how to verify many triples with just one call to the open procedure. The idea behind this improvement is straightforward: compute a random linear combination of many $[v]$'s and then open the result. If any $[v] \neq 0$, then since the coefficients are random, the final result will not equal 0, except with probability $1/(|\mathbb{F}| - 1)$. This is specified in Protocol 3.8. Observe that this method also applies to Protocol 3.4, thereby reducing the number of openings there from 3 to 2. From an asymptotic view, this is less significant, as the overhead of the protocol still depends on the cost of the open procedure.

PROTOCOL 3.8 (Batch Verification of Triples based on Semi-Honest Multiplication).

- **Inputs:** The parties hold a list of triples $\{([x_i], [y_i], [z_i])\}_{i=1}^{L}$ to verify and a list of random triples $\{([a_i], [b_i], [c_i])\}_{i=1}^{L}$.
- **The protocol:**
  1. For $i = 1$ to L: The parties run Steps 1-7 of Protocol 3.6 on $([x_i], [y_i], [z_i])$ and $([a_i], [b_i], [c_i])$, to obtain $[v_i]$.
  2. The parties call $\mathcal{F}_{\text{coin}}$ to receive random elements $\rho_1, \ldots, \rho_L \in \mathbb{F} \setminus \{0\}$
  3. The parties locally compute
  $$[v] = \rho_1 \cdot [v_1] + \ldots + \rho_L \cdot [v_L]$$
  4. The parties run open$([v])$.
  5. If no abort message was received, then the parties output accept.

Lemma 3.9. *Let $\{([a_i], [b_i], [c_i])\}_{i=1}^{L}$ be valid shares and let π_{mult} be a protocol that is secure up to additive attack. If $\{([x_i], [y_i])\}_{i=1}^{L}$ are correct, but there exists some $k \in [L]$ such that $[z_k]$ is valid and $([x_k], [y_k], [z_k])$ is not a correct multiplication triple, then the honest parties output* accept *in Protocol 3.8 with probability at most negligibly greater than $\frac{1}{|\mathbb{F}|-1}$.*

Proof. Assume that $\{([x_i], [y_i])\}_{i=1}^{L}$ are correct, and that $\exists k \in [L]$ such that $[z_k]$ is valid and $([x_k], [y_k], [z_k])$ is not a correct multiplication triple. If the honest parties output accept at the end of the protocol, then by the properties of the open procedure, $[v]$ is a correct sharing of 0. There are three cases to consider:

1. $[v_k]$ *is a correct sharing of 0.* By Lemma 3.7, the probability that this happens is at most negligibly greater than $\frac{1}{\mathbb{F}} < \frac{1}{|\mathbb{F}|-1}$.
2. $[v_k]$ *is an incorrect sharing.* Then, by Lemma 2.4, the probability that $[v]$ is a correct sharing is at most $\frac{1}{|\mathbb{F}|-1}$.
3. $[v_k]$ *is a correct sharing of some* $d_k \in \mathbb{F} \backslash \{0\}$. Let $[u] = \sum_{j \in [L] \setminus \{k\}} \rho_j \cdot [v_j]$. Then, we have that $[v] = [u] + \rho_k [v_k]$. Since $[v]$ and $[v_k]$ are both correct, it follows from the linear property of the scheme that $[u]$ is also correct. Thus, if $v = 0$ then it must hold that $0 = u + \rho_k \cdot d_k$, which in turn holds only if $\rho_k = -u \cdot d_k^{-1}$. Since $\rho_k \in \mathbb{F} \setminus \{0\}$ is random, this happens with probability of at most $\frac{1}{|\mathbb{F}|-1}$.

Thus, in all cases, the honest parties accept with probability at most $\frac{1}{|\mathbb{F}|-1}$, as required. □

4 THE PROTOCOL FRAMEWORK FOR LARGE FIELDS

In this section, we present our protocol for large fields (the protocol works for any field, but as we will see it is most efficient for large fields). The protocol has a set-up phase to generate random triples and an online phase to compute any arithmetic circuit.

The set-up protocol is presented in Protocol 4.1. In this protocol, we utilize the fact that the random triples $([a_i], [b_i], [c_i])$ used in the verification protocols only need to be valid (but not necessarily correct). This enables us to generate the tuples very efficiently, using a private semi-honest multiplication protocol.

PROTOCOL 4.1 (Generating Random Multiplication Triples).

Let π_{mult} be a private semi-honest multiplication protocol.

- **Inputs:** The parties have the number N of triples to generate.
- **The protocol:**
  1. The parties call $\mathcal{F}_{\text{rand}}$ to obtain $2N$ random sharings, arranged in a list of the form $\{([a_i], [b_i])\}_{i=1}^{N}$.
  2. For $i = 1$ to N: the parties execute π_{mult} on $[a_i]$ and $[b_i]$ to obtain $[c_i]$.
- **Outputs:** The parties output $\{([a_i], [b_i], [c_i])\}_{i=1}^{N}$.

We now proceed to the main protocol, that computes an arithmetic circuit on the private inputs of the parties. The protocol works by computing the circuit using a private semi-honest protocol, and then running a verification step where the computations of all multiplication gates are verified using the random triples from the offline phase. A full description appears in Protocol 4.2. Note that the verification stage of the protocol has two versions, as we have two protocols for verifying triples: the first uses share opening whereas the second uses semi-honest multiplication. If the second verification protocol is used, then π_{mult} also needs to be secure up to an additive attack.

We now prove the security of the protocol. The proof follow a straightforward simulation strategy, with the simulator providing shares of random values throughout (except for the output phase). The fact that the simulation works is due to the proofs already carried out that the adversary can cheat with only negligible probability.

Theorem 4.3. *Let f be a n-party functionality and let π_{mult} be a private semi-honest multiplication protocol (if VERSION 2 is used: that is secure up to additive attack). Then, Protocol 4.2 securely computes f with abort in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{rand}})$-hybrid model with statistical error $2^{-\sigma}$, in the presence of a malicious adversary controlling $t < \frac{n}{2}$ parties.*

Proof Sketch. We will show that if the adversary cheats it succeeds with probability of at most $p = (\frac{1}{|\mathbb{F}|-1})^{\delta}$. Specifically, if it provides incorrect shares in the "input sharing" step of the online protocol, then by Lemma 3.2, it is not caught with probability at most p (it is undetected with probability $1/(|\mathbb{F}| - 1)$ in each of the δ iterations). If it cheats in the multiplication gate computation in the circuit emulation phase, then regardless of the version of verification step used, it is also caught with probability p, by Lemma 3.5 and by Lemma 3.9. Note that in both cases, this holds regardless of whether or not the triples obtained from the offline protocol are correct.

Denote by *bad*, the event that the adversary succeeds in cheating without being caught, by either dealing incorrect shares for its inputs or by cheating in the computation of at least one multiplication gate. Then, $\Pr[bad] \le \left(\frac{1}{|\mathbb{F}|-1}\right)^{\delta}$. Since $\delta \cdot \log(|\mathbb{F}| - 1) \ge \sigma$, it follows that $(|\mathbb{F}| - 1)^{\delta} \ge 2^{\sigma}$ and so $\left(\frac{1}{|\mathbb{F}|-1}\right)^{\delta} \le 2^{-\sigma}$. Thus, $\Pr[bad] \le 2^{-\sigma}$.

We now show to simulate the protocol; the simulation will be "good" assuming that the *bad* event does not occur. Let $\mathcal{A}$ be the adversary. The simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ extracts the (correct) input values v shared by $\mathcal{A}$ for the corrupted parties, and hands $\mathcal{A}$ the honest parties' messages computed using 0 as the input on every wire associated with an honest party's input.
2. $\mathcal{S}$ simulates the input correctness check by running the honest parties' instructions based on the messages from the previous round. If any of the input shares dealt by the adversary is not correct, but the honest parties' simulated by $\mathcal{S}$ do not catch $\mathcal{A}$ in any iteration, then $\mathcal{S}$ outputs fail. Otherwise, $\mathcal{S}$ simulates the honest parties aborting at the relevant point, sends abort to the trusted party, and outputs whatever $\mathcal{A}$ outputs.
3. $\mathcal{S}$ runs the honest parties' instructions as above in the circuit emulation phase. If $\mathcal{A}$ cheats in any of the multiplication protocols, then $\mathcal{S}$ records in which multiplications cheating took place ($\mathcal{S}$ knows the values on the wires and so can determine if cheating took place).
4. $\mathcal{S}$ runs the honest parties' instructions in the verification stage. If there is a multiplication that $\mathcal{A}$ cheated in, but was not detected at any point by the honest parties in the simulated verification, then $\mathcal{S}$ outputs fail and halts. Otherwise, $\mathcal{S}$ simulates the honest parties aborting at the relevant point, sends abort to the trusted party, and outputs whatever $\mathcal{A}$ outputs.
5. If there was no cheating, then $\mathcal{S}$ sends the trusted party the extracted input values, and receives back the corrupted parties' outputs. Then, $\mathcal{S}$ simulates the reconstruction so that $\mathcal{A}$ receives these outputs for all corrupted parties. This is achieved by choosing "new" shares for the honest parties such that the reconstruct procedure will yield the correct output.
6. $\mathcal{S}$ receives the messages from $\mathcal{A}$ for the reconstructions to the honest parties. If any of the messages for honest P_j are incorrect (.i.e., the shares are not correct), then $\mathcal{S}$ sends abort_j to instruct the trusted party to not send the output to P_j. Otherwise, $\mathcal{S}$ sends continue_j to the trusted party, instructing it to send P_j its output.

Observe that the probability that $\mathcal{S}$ outputs fail is exactly the probability that the event *bad* occurs. Furthermore, when this does *not* occur, the only difference between the real and simulated executions is due to the fact that $\mathcal{S}$ used 0 for all honest inputs instead of the real values. By the information-theoretic security of secret sharing, this makes no difference in the input sharing phase. Furthermore, by the privacy of the multiplication protocol, the view of $\mathcal{A}$ in the simulated circuit emulation phase is indistinguishable

PROTOCOL 4.2 (Computing an Arithmetic Circuit Over Finite Fields).

Let π_{mult} be private semi-honest multiplication protocol. If VERSION 2 is used, then π_{mult} must also be secure up to additive attack.

- **Inputs:** Each party P_j ($j \in \{1, \ldots, n\}$) holds an input $x_j \in \mathbb{F}^{\ell}$.
- **Auxiliary Input:** The parties hold a description of an arithmetic circuit C that computes f on inputs of length $\ell \cdot n$. Let N be the number of multiplication gates in C. In addition, the parties hold a statistical security parameter σ.
- **The protocol:**
  1. *Precomputation:* Each party sets δ to be the smallest value for which $\delta \geq \sigma / \log(|\mathbb{F}| - 1)$. The parties then run δ executions of Protocol 4.1 with input N, and obtain vectors $\vec{d}_1, \ldots, \vec{d}_\delta$ of N triples.
  2. *Sharing the inputs:* For each input wire with an input v, the parties run share(v) with the dealer being the party whose input is associated with that wire.
  3. *Correctness checking of inputs:* Let $[v_1], \ldots, [v_m]$ be the shares on the input wires, generated in the previous step.
     Repeat δ times: The parties run Protocol 3.1 on $[v_1], \ldots, [v_m]$.
     If there exists an execution in which a party did not output accept, it sends $\perp$ to the other parties and halt.
  4. *Circuit emulation:* Let $G_1, ..., G_L$ be a predetermined topological ordering of the gates of the circuit. For $k = 1, ..., L$ the parties work as follows:
     - *If G_k is an addition gate:* Given shares $[x]$ and $[y]$ on the input wires, the parties locally compute $[x + y]$.
     - *If G_k is a multiplication-by-a-constant gate:* Given share $[x]$ on the input wire and a public constant $a \in \mathbb{F}$, the parties locally compute $[a \cdot x]$.
     - *If G_k is a multiplication gate:* Given shares $[x]$ and $[y]$ on the input wires, the parties run π_{mult} on $[x]$ and $[y]$, and define the result as their share on the output wire.
  5. *Verification stage: Before* the secrets on the output wires are reconstructed, the parties verify that all the multiplications were carried out correctly, as follows. Let $\{([x_k], [y_k], [z_k])\}_{k=1}^{N}$ be the triples generated by computing multiplication gates (i.e., $[x_k]$ and $[y_k]$ are the shares on the input wires of the kth multiplication gate and $[z_k]$ is the share on the output wire), and let $\vec{d}_i = \left\{([a_k^i], [b_k^i], [c_k^i])\right\}_{k=1}^{N}$ be the triples generated in ith iteration of the offline phase.
     For $i = 1$ to δ, the parties work as follows:
     - **VERSION 1**:
       For $k = 1, \ldots, N$: The parties run Protocol 3.4 on input $([x_k], [y_k], [z_k])$ and $([a_k^i], [b_k^i], [c_k^i])$ to verify $([x_k], [y_k], [z_k])$. (Observe that all executions of Protocol 3.4 can be run in parallel).
     - **VERSION 2**:
       The parties run Protocol 3.8 on $\{([x_k], [y_k], [z_k])\}_{k=1}^{N}$ and $\left\{([a_k^i], [b_k^i], [c_k^i])\right\}_{k=1}^{N}$ to verify $\{([x_k], [y_k], [z_k])\}_{k=1}^{N}$.

     If a party did not output accept in every execution, it sends $\perp$ to the other parties and outputs $\perp$.
  6. If any party received $\perp$ in any of the previous steps, then it outputs $\perp$ and halts.
  7. *Output reconstruction:* For each output wire of the circuit, the parties run reconstruct$([v], j)$, where $[v]$ is the sharing of the value on the output wire, and P_j is the party whose output is on the wire.
  8. If a party received $\perp$ in any call to the reconstruct procedure, 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.

from its view in the real phase. Finally, the output phase is once again perfectly simulated, by the information-theoretic properties of the secret sharing scheme. □

Using pseudo-randomness to reduce the number of calls to $\mathcal{F}_{\text{coin}}$. Observe that in some phases, we need to generate many random elements at once. Instead of calling $\mathcal{F}_{\text{coin}}$ for every value, it suffices to call it once to obtain a seed for a pseudorandom generator, and then each party locally uses the seed to obtain as much randomness as needed. (Practically, the key would be an AES key, and randomness is obtained by running AES in counter mode.) It is not difficult to show that by the pseudorandomness assumption, the probability that the adversary can cheat is only negligibly different.[1]

Efficiency. We analyze the performance of our protocol. As the protocol takes black-box building blocks (such as $\mathcal{F}_{\text{rand}}$, π_{mult} and the open procedure), and compiles them into a maliciously secure protocol, we measure the cost as a function of these building blocks. In addition, we focus on the cost that increases with the size of the circuit, and so the cost of input sharing and output reconstruction is ignored (these steps are very cheap in practice anyway). Since addition and multiplication-by-a-constant gates are computed locally, this reduces to measuring the cost per multiplication gate.

For computing multiplication gates, the protocol requires one execution of a semi-honest multiplication when emulating the circuit. In addition, for each gate, δ random multiplication triples are generated in the offline phase. This comes at the cost of generating two random shares using $\mathcal{F}_{\text{rand}}$ and multiplying them using one semi-honest multiplication.

Next, for the first version of the verification step, three opening of shares. Thus, overall, the cost of version 1 of the protocol per

[1] Note that this is not as immediate as it seems since the adversary has the seed/key as well, and so at this point the pseudorandom property is actually lost. However, the checks work by generating the randomness after everything else is finished (see Protocol 3.8) and then verifying that some equality holds, or that the results are correct. These properties are actually determined *before* the key is revealed, and thus security is maintained even after the key is revealed.

each multiplication gate is

$$(1+\delta)\cdot t(\pi_{\text{mult}})+2\delta\cdot t(\mathcal{F}_{\text{rand}})+3\delta\cdot t(\text{open}) \tag{7}$$

where $t(X)$ denotes the cost of running procedure X.

In contrast, the second version of the verification step, requires 4 semi-honest multiplications in each iteration plus one addition call to $\mathcal{F}_{\text{rand}}$ to generate $[\alpha]$, instead of the three openings. Thus, the overall cost per multiplication gate in this protocol version is

$$(1+5\delta)\cdot t(\pi_{\text{mult}})+3\delta\cdot t(\mathcal{F}_{\text{rand}}). \tag{8}$$

Observe that when using a field $\mathbb{F}$ with $|\mathbb{F}|>2^{\sigma}$ (e.g., $\mathbb{F}=\mathbb{Z}_p$ with p being a 40-bit prime, and where the allowed statistic error is 2^{-40}), it suffices to take $\delta=1$. In this case, the cost of the first version of the protocol is $2\cdot t(\pi_{\text{mult}})+2\cdot t(\mathcal{F}_{\text{rand}})+3\cdot t(\text{open})$ and the cost of the second version is $6\cdot t(\pi_{\text{mult}})+3\cdot t(\mathcal{F}_{\text{rand}})$. Thus, when 4 semi-honest multiplications plus one random-share generation are cheaper than 3 openings, the second protocol version is preferable.

5 THE PROTOCOL FOR SMALL FIELDS

The efficiency of the protocol described in Section 4 depends on the value δ. Since we require that $\delta\geq\sigma/\log(|\mathbb{F}|-1)$ for a given statistical parameter σ, this implies that δ will be large for small fields, and this will result in a far less efficient protocol. This is especially the case since the verification step is by far the most expensive part of the protocol, and this is repeated δ times. For this reason, it is desirable to use a different verification method when the field is small.

In the full version of this paper, we describe a protocol for generating multiplication triples in small fields, that combines the verification method of Section 4 together with the "cut-and-choose" methodology designed for Boolean circuits in [19]. Informally speaking, the parties start by generating $NB+C$ random triples by calling $\mathcal{F}_{\text{rand}}$ and running the semi-honest multiplication protocol π_{mult} for each triple. Next, the parties randomly permute the triples. Then, the parties open the first C triples, so that if one of the opened triples is incorrect, the honest parties will detect it and abort. The remaining triples are divided into N buckets of size B, and the first triple in each bucket is verified using the other $B-1$ triples. The required property in this check is that if one of the bucket is "mixed", i.e., contains both correct and incorrect triples, then the honest parties will detect cheating with probability 1. The important observation is that this verification of triples in a bucket can be carried out using Protocol 3.4 (verification based on openings), since by Lemma 3.5 if an incorrect triple is verified by a correct triple, then the honest parties will abort with probability 1. However, the result is much stronger than that achieved in [19]. Specifically, when considering Boolean circuits, the adversary can evade detection if a bucket contains only incorrect triples. However, in our setting where $|\mathbb{F}|>2$, even if a bucket has only bad triples, the adversary is still caught with probability $1/(|\mathbb{F}|-1)$. Thus, the cheating probability of the adversary is much lower for fields that are not too small.

We note that for the case of Boolean circuits, our protocol is essentially that of [1]. However, for small fields like $GF[2^8]$, for example, it is a hybrid of techniques from [1] and Section 4. It is worth noting that $GF[2^8]$ can be used to securely compute *Boolean* circuits, for any $n<256$ parties.

6 INSTANTIATIONS

Our protocol/compiler is generic and can be instantiated in many ways (with different secret sharing schemes, multiplication protocols, and more). Clearly, the efficiency of our protocol depends significantly on the instantiations. In this section, we present two main instantiations of our protocol, with different options for some of the subprotocols within. The first instantiation is for the general case of any number of parties n, and we use Shamir's secret sharing [35] for this instantiation. We provide different approaches to implementing the basic building blocks, including the open procedure, randomness generation $\mathcal{F}_{\text{rand}}$, and semi-honest multiplication π_{mult}, and analyze their efficiency. The second instantiation is for the specific case of three-parties, and utilizes tools from the highly efficient semi-honest protocol of [3]. Using our compiler, we show that it is possible to obtain a protocol that is secure in the presence of a malicious adversary with very low communication; specifically, only a few field elements are sent for each multiplication gate. For simplicity, we analyze our instantiations using the protocol of Section 4 for large fields only; similar analysis using the protocol for small fields of Section 5 can be easily obtained.

6.1 Multi-Party Computation Based on Shamir's Secret Sharing Scheme

Most secure computation protocols with an honest majority use Shamir's secret sharing scheme [35]. In this scheme, a secret is distributed among n parties with a threshold of t, by defining a polynomial of degree t, and handing each party a point on this polynomial. Formally, given a secret v, the dealer chooses random coefficients $q_1,\ldots,q_t\in\mathbb{F}$ and defines a polynomial $q(x)=v+\sum_{j=1}^{j=t}q_jx^j$. Then, each party P_i is given the value $q(i)$.[2] It is well known that no subset of t parties can compute the secret by themselves, but a subset of $t+1$ parties can compute the secret, as any $t+1$ points uniquely define one polynomial of degree t.

For this secret sharing scheme, a sharing $[v]$ is *correct* if there exists a single polynomial $q(x)$ of degree $\leq t$ such that for every honest party P_i holding value v_i, it holds that $q(i)=v_i$. Therefore, a sharing $[v]$ is *incorrect* if for every polynomial $q(x)$ of degree at most t, there exists an honest party P_i holding v_i such that $q(i)\neq v_i$. Note that for this scheme, if the sharing $[v]$ is not correct, then it is value-inconsistent (see Section 2). This holds since any subset of $t+1$ shares defines a t-degree polynomial, and thus for any subset of $t+1$ honest parties J it holds that $\mathsf{val}([v])_J\neq\bot$. (Technically, for this to work, all honest parties must hold some share in the field $\mathbb{F}$. Thus, if an honest party receives a value in any protocol that does not define a value in $\mathbb{F}$, it replaces it with some default field value.)

Shamir's secret sharing scheme is linear, and enable parties locally add shares and multiply them by a constant, as required for our protocol.

6.1.1 *The Basic Procedures and Sub-Protocols.*

The share(v) *procedure.* As explained above, in this procedure the dealer chooses a random polynomial $q(x)$ of degree t under the constraint that $q(0)=v$, and then sends each P_i the point

[2] More generically, we associate a unique $\alpha_i\in\mathbb{F}$ for the ith party. For simplicity, we will consider $\mathbb{F}=\mathbb{Z}_p$ for a prime p, in which case $i\in\mathbb{F}$.

$q(i)$. Recall that our protocol does not require the share sent by the corrupted parties to be correct, and thus this is sufficient (in places where correctness is required, we run a separate check).

The reconstruct$([v], i)$ *and* open$([v])$ *procedures.* In order to reconstruct to party P_i, each party sends its share to P_i. Then, P_i uses any of the $t + 1$ shares to compute the unique degree-t polynomial defined by the points, and checks that all other shares lie on the same polynomial. If not, then it sends $\perp$ to all the other parties and halts. Since there are at least $t + 1$ honest parties (whose shares uniquely define a polynomial and so the value to be reconstructed), we are guaranteed that the corrupted parties cannot cause P_i to output an incorrect value.

The open procedure simply works by running reconstruct$([v], i)$ for all P_i. Since each party sends $n - 1$ elements, the overall complexity of the open procedure is quadratic in the number of parties.

$\mathcal{F}_{\text{rand}}$ *- generating shares of random values.* We describe two ways of implementing $\mathcal{F}_{\text{rand}}$. The first method of generating random shares is the PRSS method of [13], which enables the parties to generate a sharing of a pseudorandom element without any interaction. This is done by distributing random keys among the parties via replicated secret sharing (i.e., for each subset of t parties, all the parties in the *complement* subset receive a key). These random keys are used to generate pseudorandom values that are then converted to Shamir shares. Since this protocol is non-interactive, it is easy to see that generated shares are *always correct*, and the adversary cannot learn anything about the shared value. Thus, this protocol securely realizes $\mathcal{F}_{\text{rand}}$. The problem with this approach is that the number of keys held by each party grows exponentially in the number of parties, which dramatically increases the computational work of generating the shares. Thus, this method is only efficient when the number of parties in the protocol is small. A full description of this method appears in Appendix A.1 (Protocol A.1).

We now describe a second way of generating random shares due to [17], in which each party sends only a constant number of elements per random share. This method uses a Vandermonde matrix, which can be used to "extract randomness" from n shares into $n - t$ new shares. The protocol works by having each party share a random element to the other parties. Then, upon holding a vector of n shares, each party locally multiplies this vector of size n with a Vandermonde matrix of $n - t$ rows and n columns to receive a vector of $n - t$ "new" random shares. By the randomness extraction property, we have that the new shares are sharings of random elements in $\mathbb{F}$. Since $t < \frac{n}{2}$, we have that each party obtain *at least* $\frac{n}{2} + 1$ shares, in a process that requires sending $n - 1$ elements by each party. Thus, the amortized communication cost per random share is roughly 2 elements per party. A full description is in Appendix A.1 (Protocol A.2); for more details on the method see [17]. We stress that although this method guarantees that the shares generated are to random values unknown to the adversary, the method actually does *not* securely realize $\mathcal{F}_{\text{rand}}$, as malicious parties may cheat and cause the resulting sharing to be incorrect. However, the requirements on the shares $[a]$, $[b]$ in Lemmas 3.7 and 3.9 are merely that they be valid, and all Shamir shares are valid as we have mentioned. (Note that in Lemma 3.5, this is not even needed.) Thus, this method suffices for our protocol.

π_{mult} *– semi-honest multiplication that is private and secure up to additive attack.* The first instantiation for π_{mult} is the GRR multiplication improvement [25] of the BGW protocol [7]. This protocol enables n parties with $t < \frac{n}{2}$ semi-honest corrupted parties, to compute a multiplication gate with quadratic communication complexity overall. Specifically, it works by each party locally multiplying its shares on the input wires and sending shares of the result to all other parties. Then, each party locally computes a linear combination of the shares it received. The overall communication is thus exactly $n - 1$ elements sent by each party per multiplication gate. This protocol is very efficient for a small number of parties.

The second instantiation is more efficient for a large number of parties. The semi-honest multiplication protocol with the best asymptotic efficiency to our knowledge is the DN protocol presented in [17], that has amortized constant communication complexity per party. We optimized the protocol even further, cutting the communication by half. The difference is due to the fact that [17] generate correct multiplication triples and use them to carry out the multiplications in the circuit. In contrast, we generate just one share (under degree-t and degree-$2t$) and use this to multiply directly. Their method is preferable for minimizing the online time, but ours yields a faster overall time. Our optimized version of the [17] protocol works as follows. A set of random shares is generated in a preprocessing step, and later used to multiply two elements that are shared among the parties. Specifically, the parties generate two random shares $[r]_t$ and $[r]_{2t}$ for each multiplication gate, where the former is a sharing of r using a degree-t polynomial, and the latter is a sharing of the same r, but this time using a degree-$2t$ polynomial. Then, to multiply x and y, the parties locally multiply their shares to obtain $[x \cdot y]_{2t}$ (the result is of degree-$2t$ since two degree-t polynomials are multiplied). Then, the parties locally compute $[x \cdot y]_{2t} - [r]_{2t}$ and open the result to P_1. Party P_1 obtains $x \cdot y - r$ in the clear, and sends it to all the other parties. Finally, each party locally computes $x \cdot y - r + [r]_t$ to obtain $[x \cdot y]_t$, as required (the result is a degree-t polynomial with constant term $x \cdot y$ since the constant term of $[r] - r$ is 0). If we use the "Vandermonde" protocol for generating the random shares, we obtain that each party needs to send only 6 field elements per multiplication gate.

Both semi-honest multiplication protocols are presented in Appendix A.2 (Protocol A.3 and Protocol A.4). We claim that these two protocols prevent any leakage of information, and are private in the presence of malicious adversaries according to Definition 2.5. For the GRR protocol, this follows from the fact that the adversary's view consists of random shares it receives from the honest parties, which reveal nothing. For the protocol of [17], the view of the adversary consists of random shares or of values that are a masking of some secret using a random value, thus once again revealing nothing about the actual values.

In instantiations that rely on multiplication-based verification, we will only use the DN protcol, as these instantiations yield higher performance when the computation involves large number of parties, and the DN protocol, with its constant communication per party, is clearly preferable in this case over the GRR protocol. Thus, it is required that this protocol be secure up to additive attack. Fortunately, it was already proven in [21, Corollary 5.6] that the DN protocol has this property.

6.1.2 Protocol Variants. We have described two protocols for generating random shares and two semi-honest multiplication protocols. Together with the two versions of our protocol (verification via opening in Section 3.3 and via multiplication in Section 3.4), we have six variants of our multiparty protocol with Shamir's secret sharing (recall that we did not prove that the GRR protocol can be used with the multiplication-based verification). We analyze two of these variants; one is best for a small number of parties and the other is best for a large number of parties.

A protocol for a small number of parties. When the number of parties is small, we use the PRSS protocol to generate random shares (with zero communication cost), the GRR multiplication protocol (with $n-1$ elements sent per party) and version 1 of our protocol (which uses verification based on opening of shares). The reason why we use verification via opening of shares is that the open procedure is very efficient for a small number of parties and thus is preferable here (here, each party sends $n-1$ elements for both multiplication and opening).

For large fields where $|\mathbb{F}| > 2^{-\sigma}$ we can set $\delta = 1$. In this case, referring to Eq. (7), we obtain that the overall communication cost is $5(n-1)$ *field elements* sent by each party for each multiplication gate (due to 2 semi-honest multiplications and 3 openings). This therefore proves Theorem 1.2 from the Introduction.

A protocol for a large number of parties. When considering a large number of parties, the PRSS protocol cannot be used (since the computational cost of the PRSS protocol blows up exponentially). We therefore use the "Vandermonde" protocol for generating random shares. Likewise, we use the optimized DN [17] protocol for semi-honest multiplication and the second version of our protocol (which is based on semi-honest multiplications). For large fields, where we can set $\delta = 1$, we obtain that by Eq. (8), each party needs to send 42 *elements* per multiplication gate (due to 6 semi-honest multiplications at a cost of 6 elements each, and 3 random share generations at a cost of 2 elements each). This therefore proves Theorem 1.1 from the Introduction.

The threshold. Based on the above, when $5(n-1) < 42$ the first protocol is better. Thus, the first protocol should perform better when the number of parties is $n \leq 9$, and the second protocol should perform better when $n > 9$. This theoretical analysis is validated experimentally in Section 7 and confirms this exactly. In Table 1, the protocol labeled PRSS_GRR_open is the first protocol, whereas van_DN_mult is the last protocol. As can be seen, PRSS_GRR_open is better than van_DN_mult for up to 9 parties. As shown in Section 7, other protocol variants are however better between 9 and 70 parties.

6.2 Three-Party Computation Based on Replicated Secret Sharing

In [3], a three-party protocol based on a type of replicated secret sharing was presented, in which each party sends a single field element per multiplication gate. In this section, we instantiate our protocol compiler with the semi-honest protocol of [3] (for arithmetic circuits) to obtain a highly efficient protocol with security in the presence of a malicious adversary corrupting at most 1 party. We stress that protocols that require that less than 1/3 of the parties are corrupted cannot be used at all for the case of 3 parties.

We begin by presenting the replicated secret-sharing scheme, prove some properties of it, and characterize correctness. Our replicated secret-sharing is a simplified version of that presented in [3], and the multiplication protocol has the same complexity as theirs. We also show how to optimize Protocol 4.2 even further in this specific case.

We stress that for Boolean circuits (i.e., the field $\mathbb{F}_2$), a malicious version for the semi-honest protocol of [3], was already presented in [19]. However, our construction for large fields is fundamentally different from theirs, as it is tailored for large fields.

6.2.1 The Secret Sharing Scheme and Its Properties.

Replicated secret-sharing. In order to share an element $v \in \mathbb{F}$, the dealer chooses three random elements $r_1, r_2, r_3 \in \mathbb{F}$ under the constraint that $r_1 + r_2 + r_3 = v$. Then, the dealer shares the secret so that P_1's share is (r_1, r_3), P_2's share is (r_2, r_1) and P_3's share is (r_3, r_2). We abuse notation and denote P_i's share by (r_i, r_{i-1}), even for $i = 1$. It is easy to see that this is a valid secret sharing scheme that preserves privacy. In addition, the secret together with the share of any one party fully determines the shares of the other parties. We use $[v]$ to denote a sharing of v according to this scheme.

Linearity. We define the following local *operations* on shares:

- *Addition* $[v_1] + [v_2]$*:* Given a share (r_i^1, r_{i-1}^1) of v_1 and a share (r_i^2, r_{i-1}^2) of v_2, the sum of the shares is obtained by each party P_i computing: $(r_i^1 + r_i^2, r_{i-1}^1 + r_{i-1}^2)$.
- *Multiplication by a scalar* $\sigma \cdot [v]$*:* Given a share (r_i^1, r_{i-1}^1) of v and a value $\sigma \in \mathbb{F}$, each party P_i computes $(\sigma \cdot r_i, \sigma \cdot r_{i-1})$.
- *Addition of a scalar* $[v] + \sigma$*:* Given a share (r_i, r_{i-1}) of v and a value $\sigma \in \mathbb{F}$, party P_1 computes $(r_1 + \sigma, r_3)$, party P_2 computes $(r_2, r_1 + \sigma)$, and party P_3 leaves its share as is.

Note that when writing $[v_1] + [v_2]$ the symbol "+" is an operator on shares and not addition of two numbers, whereas when we write $v_1 + v_2$ the symbol "+" is addition in the field; likewise for the product notation. The following claim states that these operators are correct, and is straightforward to prove.

Claim 6.1. *Let* $[v_1], [v_2]$ *be shares and let* $\sigma \in \mathbb{F}$ *be a scalar. Then:*

(1) $[v_1] + [v_2] = [v_1 + v_2]$,
(2) $\sigma \cdot [v_1] = [\sigma \cdot v_1]$, *and*
(3) $[v_1] + \sigma = [v_1 + \sigma]$.

Correctness. In the three parties setting, it is not possible to have shares that are value-inconsistent, since there is only one ever subset of $t+1$ honest parties. However, the sharing might be invalid. This is the exact opposite situation to Shamir shares which are always valid but may be value-inconsistent when $n > 3$.

It will be useful to characterize the correctness of shares in this case. Consider the case that P_1 is corrupted. Then, P_2 is supposed to hold (r_2, r_1) and P_3 is supposed to hold (r_3, r_2). Thus, the sharing is valid (and thus correct) if and only if the first element held by P_2 equals the second element held by P_3. In general:

Claim 6.2. *Let* $(r_1, s_1), (r_2, s_2)$ *and* (r_3, s_3) *be the shares held by parties* P_1, P_2 *and* P_3*, respectively, and let* P_i *be the corrupted party. Then, the shares are* correct *if and only if* $r_{i+1} = s_{i+2}$.

6.2.2 *Basic Building Blocks and Sub-Protocols.*

$\mathcal{F}_{\text{rand}}$ - *Generating shares of random values.* The parties can generate shares of random values non-interactively in the following way. Let $F_k(\cdot)$ be a pseudorandom function. Then:

- *Initialization*: Each party P_i chooses a random key k_i and sends it to P_{i+1}. Each party initializes a counter $id = 0$.
- *Share generation*: Upon each request to generate a sharing, each party P_i holding two keys k_i and k_{i-1} sets $id = id + 1$ and computes $r_{i-1} = F_{k_{i-1}}(id)$ and $r_i = F_{k_i}(id)$. Then, P_i outputs the share (r_i, r_{i-1}).

Note that the random sharing output from the protocol is guaranteed to be correct, since the protocol requires no communication. It is not difficult to show that this protocol securely realizes $\mathcal{F}_{\text{rand}}$.

π_{mult} – *Semi-Honest Multiplication Protocol.* We describe the multiplication protocol in which each party sends only one field element. Let (r_1, r_3), (r_2, r_1), (r_3, r_2) be a secret sharing of v_1, and let (s_1, s_3), (s_2, s_1),(s_3, s_2) be a secret sharing of v_2. We assume that the parties P_1, P_2, P_3 hold *correlated randomness* $\alpha_1, \alpha_2, \alpha_3$, respectively, where $\alpha_1 + \alpha_2 + \alpha_3 = 0$. The parties compute shares of $v_1 \cdot v_2$ as follows:

(1) **Step 1 – compute $\binom{3}{3}$-sharing:** Each party P_i computes $t_i = r_i s_i + r_i s_{i-1} + r_{i-1} s_i + \alpha_i$ and sends it to P_{i+1}. These messages are computed and sent in parallel.
(2) **Step 2 – compute $\binom{3}{2}$-sharing:** Party P_i computed t_i and received t_{i-1} from P_{i-1}; party P_i outputs (t_i, t_{i-1}) as its share on the output wire. (If P_i received $t_{i-1} \notin \mathbb{F}$ then it sets t_{i-1} to a default element in $\mathbb{F}$.)

Observe that $t_1 + t_2 + t_3 = \sum_{i=1}^{i=3}(r_i s_i + r_i s_{i-1} + r_{i-1} s_i) + \sum_{i=1}^{i=3} \alpha_i = v_1 \cdot v_2$, where the equality follows since $\sum_{i=1}^{3} \alpha_i = 0$ and $v_1 \cdot v_2 = (r_1 + r_2 + r_3)(s_1 + s_2 + s_3)$. Thus, when the parties are honest, the obtained sharing is a correct sharing of $v_1 \cdot v_2$. In addition, the protocol achieves privacy in the presence of a malicious adversary according to Definition 2.5, as the adversary's view consists of one element that looks random, due to the fact that it is masked using a random element α_i. We now show that the above multiplication protocol (for *semi-honest adversaries*) always yields correct shares, even when run in the presence of a *malicious adversary*. Specifically, the result is either a correct sharing of the product or of a different field element (depending on the adversary), but it is *always* correct.

Lemma 6.3. *If* $[v_1]$ *and* $[v_2]$ *are correct and* $[v_3]$ *was generated by executing the (semi-honest) multiplication protocol on* $[v_1]$ *and* $[v_2]$ *in the presence of one malicious party, then* $[v_3]$ *is a correct sharing of either* $v_1 \cdot v_2$ *or of some element* $v \in \mathbb{F}$.

Proof. If the corrupted party follows the protocol specification then $[v_3]$ is a correct sharing of $v_1 \cdot v_2$. Else, since the multiplication protocol is symmetric, assume without loss of generality that P_1 is the corrupted party. Then, the only way that P_1 can deviate from the protocol specification is by sending an incorrect element $\tilde{t}_1$ to the honest P_2 instead of t_1, and in this case P_2 will define its share to be $(t_2, \tilde{t}_1)$. Meanwhile, P_3 defines its share to be (t_3, t_2), since it receives t_2 from the honest P_2. By Claim 6.2 the shares are correct, since the first element of P_2's share equals the second element of P_3's share. Furthermore, the shares that P_2 and P_3 hold define the secret $v = \tilde{t}_1 + t_2 + t_3 \in \mathbb{F}$, as required. □

Using similar arguments as in Lemma 6.3, it can be shown that the semi-honest multiplication protocol is also secure up to additive attack. However, we won't be needing this property in this instantiation as explained below.

Generating correlated randomness non-interactively. Generating elements $\alpha_1, \alpha_2, \alpha_3 \in \mathbb{F}$ under the constraint that $\alpha_1 + \alpha_2 + \alpha_3 = 0$ can be done in an almost identical way as generating shares of random values described above. In a set-up step, each party P_i chooses a key k_i and sends it to P_{i+1}. Then, in order to generate correlated randomness, each party computes $\alpha_i = F_{k_{i-1}}(id) - F_{k_i}(id)$ using the two keys it holds (and after incrementing id); observe that $\alpha_1 + \alpha_2 + \alpha + 3 = 0$, but each party knows nothing beyond its own α. Using this method, the parties can generate all the correlated randomness needed at the cost of one exchange of keys.

The open *and* reconstruct *procedures.* We use the fact that the protocols described so far guarantee that correctness of shares is maintained even in the presence of a malicious adversary, to construct these procedures in an efficient way.

The open$([v])$ procedure (the opening of a share to all parties) works in the following way: Holding the share (r_i, r_{i-1}), each party P_i sends the element r_{i-1} to party P_{i+1}. Then, upon receiving r_{i-2} (i.e., r_{i+1}) from P_{i-1}, party P_i computes $v = r_{i+1} + r_i + r_{i-1}$. Note that this does not guarantee that the parties will hold the same correct value. However, if the sharing is correct, then we are guaranteed that one of the honest parties will hold the correct value, and thus the parties can compare their views to detect cheating. Let compareview(v_i) be a procedure where each party P_i sends v_i to P_{i+1}. Then, P_i checks that $v_i = v_{i-1}$ and aborts if not. Adding this comparison to the open procedure doubles the communication. However, in the protocol, we can reduce communication by deferring all view comparisons to the end of the protocol, and then compare a hash of all of the values to be compared throughout the execution. This method of deferring comparisons to the end is similar to the deferred MAC verification of the TinyOT and SPDZ protocols [16, 18, 32]; we use this method throughout. Thus, we conclude that the procedure requires each party to send 1 field element only per multiplication gate.

The reconstruct$([v], i)$ procedure works by having party P_{i-1} send r_{i-2} to P_i and party P_{i+1} send r_{i+1} to P_i. Then, P_i checks that $r_{i-2} = r_{i+1}$. If not, it sends $\perp$ and aborts. If yes, it computes $v = r_{i-2} + r_{i-1} + r_i$. This works since, if the corrupted party cheats and sends the wrong value, then it will be detected by P_i. As before, this procedure works only when $[v]$ is correct. If this is not guaranteed, then reconstruction can be carried out by both parties sending P_i their entire share (at double the communication cost).

The share(v) *procedure.* We define this procedure in a similar way to [19], relying on the fact that the protocol for generating random shares provides correct sharings. First, the parties generate a random sharing $[r]$. Then, they run reconstruct$([r], i)$. Holding r, the dealer P_i sends $b = v - r$ to the other parties. Finally, the parties run compareview(b) to ensure that the dealer sent them the same value (recall that in the protocol we defer the view comparisons to the end and then only send a single hash value, so this does not require communication). If no $\perp$ message was received, the parties define their share of v to be $[r] + b$ (as defined above).

Since we are guaranteed that $[r]$ is correct, then $[v] = [r] + b$ is also correct, assuming that the honest parties hold the same b as ensured by running compareview(b).

6.2.3 Optimizing the Protocol - Reducing the Communication. With the building blocks that we presented in the previous section, the protocol is now completely defined. However, we can improve the performance even further, using an optimization that is unique to the secret sharing scheme used in this instantiation.

Before proceeding, observe that the cost of the opening procedure and the semi-honest multiplication protocol are identical (one element sent per party). Thus, it is clear that in the three-party case, Protocol 3.4 (triple verification based on opening shares) is cheaper than Protocol 3.8 (triple verification based on multiplication), as the former requires 3 openings whereas the latter requires 4 multiplications per gate. We now present an optimization that reduces the number of openings in Protocol 3.4 from 3 to 2, reducing the communication per multiplication gate by 1 field element. Recall that in Protocol 3.4 the parties compute a sharing $[v]$, and then open it and verify that it is a sharing 0. In addition, recall that in the sharing scheme, each party P_i holds a pair (r_i, r_{i-1}). Thus, if $v = 0$, then $r_{i-1} + r_i + r_{i+1} = 0$ and so $r_{i-1} + r_i = -r_{i+1}$. Now, since we are guaranteed that the parties hold a correct sharing of v (the triples that are input to the multiplication protocol are correct and all operations during the protocol maintain this property), the only question that remains is whether $[v]$ is a sharing of 0 or of some other value in the field. This can be verified by having each pair of parties P_i and P_{i+1} compare the values of $r_{i-1} + r_i$ and $-r_{i+1}$. Thus, it is possible to include these values in the view comparison hash that is verified at the end of the entire protocol, instead of running open$([v])$ at each multiplication gate. This single comparison can be done efficiently by comparing the hash values of the strings that hold the shares from all Protocol 3.4 executions. Specifically, each party P_i needs to hold two strings; in the first string, it stores the $r_{i-1} + r_i$ from the $[v]$ shares it viewed, whereas in the second, it stores $-r_i$ from these shares. The hash of the first string is compared with the hash value of party P_{i+1}, and the hash of the second string is compared with the hash value of P_{i-1}.

6.2.4 Putting It All Together.

The protocol. For the three-party setting based on replicated secret sharing, we obtain a single protocol (unlike the Shamir case). First, observe that using our sharing procedure presented above, there is no need to run the input correctness-checking step. In addition, the parties run the optimized verification protocol explained in the previous section. Finally, we add an additional step before the output reconstruction, where the parties compare their views by sending each other a hash of their views. This step comes with constant small communication cost, and thus does not change the cost per gate. The full protocol is described in the full version of this paper.

Efficiency. As generation of random shares is essentially free, the cost of the protocol per multiplication gate involves two semi-honest multiplications and two openings for verification. Thus, $1 + 3\delta$ elements are sent by each party per gate, as both opening and multiplication involves sending one element per party.

In large fields, where $|\mathbb{F}| > 2^\sigma$ and $\delta = 1$, we obtain that each party needs to send *only 4 field elements* per multiplication gate. This therefore proves Theorem 1.3 from the Introduction.

7 EXPERIMENTAL RESULTS

We implemented our protocol in C++ and ran our protocols on Azure in a single region with a ping time of approximately 1ms. Each machine is a 2.4GHz Intel Xeon E5-2673 v3, with 4 cores and 8GB RAM. Each party was implemented with a single thread and so each party utilizes only on a single core.

We implemented our protocol versions and ran extensive experiments to analyze the efficiency of the different protocols for different numbers of parties. All of our protocols scale linearly in the size and depth of the circuit, and we therefore ran all of our experiments on a depth-20 arithmetic circuit with 1,000,000 multiplication gates. We ran the circuit over two different fields, one defined by a 31-bit Mersenne prime and the other defined by a 61-bit Mersenne prime. Using these two different fields is of interest since our verification protocol must be run twice ($\delta = 2$) when $|\mathbb{F}| < 2^\sigma$. That is, for security 2^{-40}, we need to take $\delta = 2$ for the 31-bit prime, and $\delta = 1$ for the 61-bit prime. Thus, this provides a tradeoff between more phases in the protocol vs working with and sending larger field elements. The protocol versions that we implemented are all for the framework for large fields as described in Section 4, and we did not implement the small field framework of Section 5.

We have described 6 possible instantiation of our Shamir-based protocol: **(1)** random share generation via PRSS [13] or using the Vandermonde method [17]; **(2)** multiplication via GRR [25] or via DN [17]; **(3)** verification via opening (Section 3.3) or via multiplication (Section 3.4). Since the verification via multiplication requires that the semi-honest multiplication protocol be secure up to additive attacks for malicious adversaries, this is only relevant for the [17] protocol. Thus, there are 6 possible options. We denote the random generation options by PRSS and van, the multiplication options by GRR and DN, and the verification options by open and mult. Using this notation, we denote by van_DN_mult the protocol that uses Vandermonde randomness generation, DN multiplication and verification via multiplication. We ran only 4 versions of the protocol. Specifically, we did not run the versions combining PRSS randomness generation and DN multiplication. This is because PRSS randomness generation works only for a small number of parties (it is exponential in the number of parties and therefore was run only for up to 11 parties) whereas DN multiplication is only better for a large number of parties. Thus, this combination is not optimal. In addition, we implemented our three-party protocol using replication sharing, from Section 6.2. We denote this protocol by replicated.

See Table 1 for the results of these protocol executions. Our results clearly validate our theoretical analysis that our 3-party protocol based on replicated sharing is the best for 3 parties, that PRSS randomness generation is best for only a very small number of parties ($n \leq 7$), that DN multiplication becomes better than GRR multiplication only for a large number of parties ($n > 11$), and that verification with multiplication is preferred for a very large number of parties ($n > 70$) since it is asymptotically linear.

Protocol version	3	5	7	9	11	30	50	70	90	110
replicated (3 party)	**513**	-	-	-	-	-	-	-	-	-
PRSS_GRR_open	1229	**1890**	**3056**	6719	18024	-	-	-	-	-
van_GRR_open	1428	2104	3214	**4009**	**5187**	20855	45902	79655	124353	177621
van_DN_open	1999	2661	3463	4426	5694	**15954**	**28978**	**44599**	63522	83815
van_DN_mult	3218	4521	5924	7279	8570	21437	34832	47379	**58966**	**72096**

Table 1: Execution time in milliseconds of the circuit with a 61-bit prime, for different numbers of parties. The best time for each number of parties is highlighted.

Figure 1 below shows the comparison of the 4 different Shamir-based protocols for 3 to 90 parties. The graph clearly demonstrates the linear complexity of the van_DN_mult protocol, and thus it is less efficient for a small number of parties but far more efficient as the number of parties grows. In contrast, the PRSS protocol increases exponentially, and the others quadratically (at different rates). In order to more closely see the behavior of the protocols for a small number of parties; see Figure 2 below.

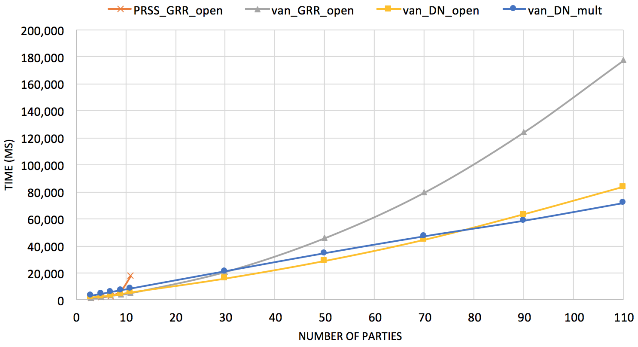

Figure 1: A comparison of the 4 different Shamir-based protocols with a 61-bit prime

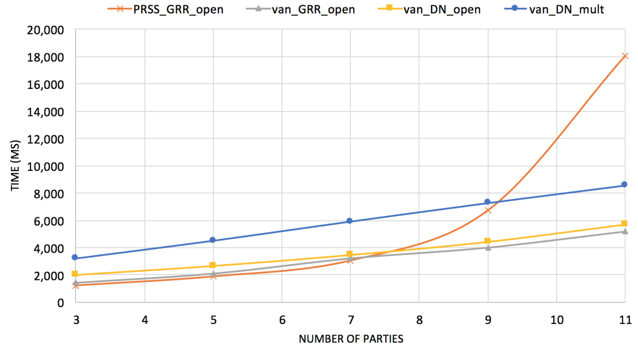

Figure 2: A comparison of the 4 different Shamir-based protocols with a 61-bit prime, for a small number of partiets

Field size. As mentioned above, we ran experiments both for the 31-bit and 61-bit fields. As can be seen in Figure 3, the running time is very similar when using a 61-bit field (and taking $\delta = 1$) versus using a 31-bit field (and taking $\delta = 2$). This makes sense because by Equations (7) and (8), the communication is almost linear in δ. Thus, the number of field elements sent when $\delta = 2$ is approximately twice the number of field elements sent when $\delta = 1$. Since the size of a single 61-bit field element is approximately twice the size of a 31-bit field element, we have that the communication is similar. Having said the above, we do observe that the protocol using a 61-bit field is slightly better for a small number of parties ($n \leq 9$), whereas the protocol using a 31-bit field is slightly better for a larger number of parties ($n \geq 11$). This can be explained by the fact that communication costs are more significant for a larger number of parties, and two times Eq. (7) with $\delta = 1$ is slightly higher than one times Eq. (7) with $\delta = 2$. Concretely, in van_DN_open the number of field elements is $t(\pi_{\text{mult}}) = 6$, $t(\mathcal{F}_{\text{rand}}) = 2$ and $t(\text{open}) = n - 1$; thus, for $n = 11$, we have that 46 group elements are sent with $\delta = 1$ and 86 group elements are sent with $\delta = 2$. Given that each group element is twice the size for $\delta = 1$, we have that this is 7% more communication.

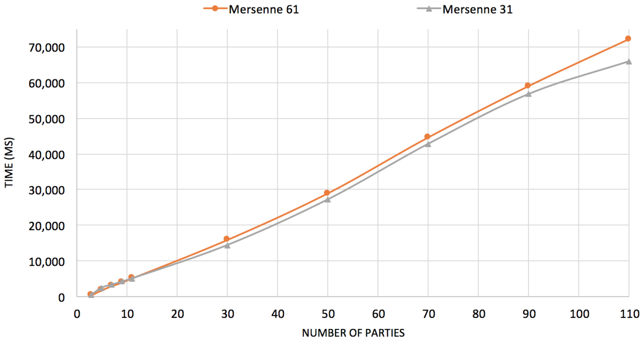

Figure 3: A comparison of the *best* running-times with 31-bit and 61-bit primes, for Shamir-based instantiations

Arithmetic or Boolean protocols. It is instructive to compare our 3-party protocol (that computes at a rate of 1,000,000 multiplications per second) to the best 3-party protocol with malicious security for Boolean circuits [1], that computes approximately 73,000,000 AND gates per second on a single core [2]. A back of the envelope calculation shows that multiplication in our protocol cost about the same as addition in a Boolean circuit (whereas addition is free here, but multiplication is very expensive in Boolean circuits). Thus, for arithmetic-based computation, our protocol is far superior (of course, in computations that require many comparisons and other types of operations, the reverse is true).

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and Mike Rosulek for helpful comments and discussion. We also thank Meital Levy and Hila Dahari for the protocol implementation, and Lior Koskas for running the experiments.

REFERENCES

[1] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman and O. Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In the *38th IEEE Symposium on Security and Privacy*, pages 843–862, 2017.

[2] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman and O. Weinstein. Personal communication, May 2017.

[3] 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.

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

[5] E. Ben-Sasson, S. Fehr and R. Ostrovsky. Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority. In *CRYPTO 2012*, Springer (LNCS 7417), pages 663-680, 2012.

[6] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC 2008*, Springer (LNCS 4948), pages 213–230, 2008.

[7] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC*, pages 1–10, 1988.

[8] 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.

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

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

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

[12] K. Chida, K. Hamada, D. Ikarashi and R. Kikuchi. Actively Private and Correct MPC Scheme in t<n/2 from Passively Secure Schemes with Small Overhead. *IACR Cryptology ePrint Archive*, report 2014/304, 2014.

[13] 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.

[14] 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.

[15] I. Damgård and Y. Ishai. Scalable Secure Multiparty Computation. In *CRYPTO 2006*, Springer (LNCS 4117), pages 501–520, 2006.

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

[17] I. Damgård and J. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO 2007*, Springer (LNCS 4622), pages 572–590, 2007.

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

[19] 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*, Springer (LNCS 10211), pages 225–255, 2017.

[20] R.A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research* (3rd ed.), pages 26–27, 1938.

[21] D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai and E. Tromer. Circuits Resilient to Additive Attacks with Applications to Secure Computation. In *STOC 2014*, pages 495-504, 2014.

[22] D. Genkin, Y. Ishai and A. Polychroniadou. Efficient Multi-party Computation: From Passive to Active Security via Secure SIMD Circuits. In *CRYPTO 2015*, Springer (LNCS 9216), pages 721–741, 2015.

[23] M. Hirt and J.B. Nielsen. Robust Multiparty Computation with Linear Communication Complexity. In *CRYPTO 2006*, Springer (LNCS 4117), pages 463–482, 2006.

[24] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game. In 19*th STOC*, pages 218–229, 1987.

[25] R. Gennaro, M. Rabin and T. Rabin. Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography. In 17*th PODC*, pages 101–111, 1998.

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

[27] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. In the *Journal of Cryptology*, 18(3):247–287, 2005.

[28] M. Keller, E. Orsini and P. Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In the *23rd ACM CCS*, pages 830–842, 2016.

[29] 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.

[30] Y. Lindell and B. Pinkas. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. In the *8th TCC*, Springer (LNCS 6597), 329–346, 2011.

[31] P. Mohassel, M. Rosulek and Y. Zhang. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *ACM Conference on Computer and Communications Security*, pages 591–602, 2015.

[32] J.B. Nielsen, P.S. Nordholt, C. Orlandi and S.S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO 2012*, Springer (LNCS 7417), pages 681–700, 2012.

[33] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT 1999*, Springer (LNCS 1592), pages 223–238, 1999.

[34] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. In 21*st STOC*, pages 73–85, 1989.

[35] A. Shamir. How to share a secret. Communications of the ACM, 22(11), pages 612–613, 1979.

[36] A. Yao. How to Generate and Exchange Secrets. In the 27*th FOCS*, pages 162–167, 1986.

A PROTOCOLS FOR COMPUTATION BASED ON SHAMIR'S SECRET SHARING SCHEME

A.1 Protocols For Generating Random Sharings

The PRSS protocol [13]. In this protocol, there is a setup step where the parties generate a random key k_A for each subset $A \subset \{P_1, \ldots, P_n\}$ of $n-t$ parties (known only to the parties in the subset). In addition, for each such subset there is a polynomial f_A of degree $t+1$ defined by the points: (1) $f_A(0) = 1$; (2) $f_A(i) = 0$ for all i such that $P_i \in \{P_1, \ldots, P_n\} \setminus A$. Then, each party uses the keys it holds to generate random shares without any interaction. The protocol is described in Protocol A.1.

PROTOCOL A.1 (The PRSS Protocol for Generating Random Shares).

Let $F_k()$ be a pseudo-random function with security parameter κ.

- **Set-up step:** For each $A \subset \{P_1, \ldots, P_n\}$, such that $|A| = n-t$, the party with the smallest lexicographic index chooses a random key k_A and sends it to all the other parties in A.
- **Upon request:** For each $A \subset \{P_1, \ldots, P_n\}$, such that $|A| = n-t$, let f_A be a $t+1$-degree polynomial defined by the points: (1) $f_A(0) = 1$; (2) $f_A(i) = 0$ for all i such that $P_i \in \{P_1, \ldots, P_n\} \setminus A$. Then, each party P_i computes
$$s_i = \sum_{A \subset \{P_1, \ldots, P_n\} : |A| = n-t,\, P_i \in A} F_{k_A}(id) \cdot f_A(i)$$
where id is public counter that is incremented for each new request, and define s_i as its share.

Batch generation of random shares using hyper-invertible matrices. The previous protocol is communication free, but its computational cost grows exponentially with the number of parties, as the number of random keys is $\binom{n}{t}$. The next protocol has linear communication and computational cost. The idea behind the protocol is to use the Vandermonde matrix to extract randomness from n random sharings into $n-t$ random sharings. Let $\gamma_1, \ldots, \gamma_n \in \mathbb{F}$ be n distinct non-zero elements. The Vandermonde matrix $\vec{V}_\ell \in \mathbb{F}^{(n,\ell)}$ is a matrix of n rows and ℓ columns defined by

$$\vec{V}_\ell \overset{\text{def}}{=} \begin{pmatrix} 1 & \gamma_1 & \cdots & \gamma_1^\ell \\ 1 & \gamma_2 & \cdots & \gamma_2^\ell \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \gamma_n & \cdots & \gamma_n^\ell \end{pmatrix}$$

We use the notation $\vec{V}_\ell^{-1}$ to denote the inverse of the Vandermonde matrix. Likewise, the transpose of the matrix is denoted by $\vec{V}_\ell^{T}$. The Vandermonde matrix has the property that if we take a subset of ℓ rows to form a square matrix of ℓ rows and ℓ columns, then the obtained matrix is guaranteed to be invertible. A matrix with this property is called "hyper-invertible". A consequence of this property is that the Vandermonde matrix can be used to extract randomness in the following way. First, each party chooses a random element and shares it to the other parties. Then, holding a vector of n shares, the parties generate "new" random shares by multiplying this vector with the Vandermonde matrix. It can be shown that the hyper-invertible property of the matrix implies that the obtained shares are sharings of a random elements in $\mathbb{F}$ [6]. The protocol is described in Protocol A.2.

PROTOCOL A.2 (Batch Generation of random shares using Vandermonde matrix).
Let $\vec{V}_{n-t}$ be the Vandermonde matrix defined above.

(1) Each party P_i chooses a random element $u_i \in \mathbb{F}$ and shares it to the other parties.
(2) Holding n shares $([u_1], \ldots, [u_n])$, each party P_i computes $([r_1], \ldots, [r_{n-t}]) = \vec{V}_{n-t}^{T} \cdot ([u_1], \ldots, [u_n])$, and defines $([r_1], \ldots, [r_{n-t}])$ as its output of the protocol.

Note that each party sends $n-1$ elements to generate $n-t$ shares. Thus, the amortized communication complexity per generated share is roughly 2, since $t < \frac{n}{2}$.

A.2 Protocols For Semi-Honest Multiplication

The GRR protocol. This protocol works by having each party locally multiply its shares of the inputs, and share the result to all the other parties. Then, upon holding n shares, each party locally computes a linear combination of the shares, and define the result as its share on the output wire. The coefficients used for the linear combination are taken from the first row of $\vec{V}_n^{-1}$, which is the inverse of the square $n \times n$ Vandermonde matrix defined above. To understand why the protocol is correct see [25]. The protocol is described in Protocol A.3.

PROTOCOL A.3 (The GRR Semi-Honest Multiplication Protocol).
Let $\vec{V}_n$ be the Vandermonde matrix defined above, and let $\lambda_1, \ldots, \lambda_n$ be the values in the first row of $\vec{V}_n^{-1}$.

- **inputs:** The parties hold sharings $[x]$ and $[y]$.
- **The protocol:**
  (1) Let f_x^i, f_y^i be the shares held by party P_i. Then, each party P_i computes $v_i = f_x^i \cdot f_y^i$ and sends shares of v_i.
  (2) Let h_i^j be the share sent from P_i to P_j in the previous step. Upon receiving shares from all the other parties, each party P_i sets its share of $x \cdot y$ to be the result of the linear combination $\sum_{j=1}^{n} \lambda_j \cdot h_j^i$.

In the protocol, each party sends exactly $n-1$ elements, and thus the overall communication complexity is quadratic in the number of parties.

The DN protocol. This next protocol, which is an optimized version of [17], requires each party to send few field elements regardless of the number of parties. The protocol has a set-up step where the parties generate two random sharings $[r]_t$ and $[r]_{2t}$ of the same value r, using t-degree and $2t$-degree polynomials in respectively. These shares are then used to multiply x and y which are shared among the parties. A full description appears in Protocol A.4.

PROTOCOL A.4 (The DN Semi-Honest Multiplication Protocol).
Let $\vec{V}_{n-t}$ be the Vandermonde matrix defined above.

- **Inputs:** The parties hold sharings $[x]$ and $[y]$.
- **Setup phase:** The parties generate a list of $n-t$ double random shares $\{[r_k]_t, [r_k]_{2t}\}_{k=1}^{n-t}$ where $[r_k]_t$ is a sharing of r_k using a t-degree polynomial and $[r_k]_{2t}$ is a sharing of r_k using a $2t$-degree polynomial. This generation works as follows:
  Each party chooses a random element u and shares it twice, using a t-degree polynomial and a $2t$-degree polynomial.
  Then, upon holding two vectors of random shares, the parties multiply each of them with $\vec{V}_{n-t}$ as described in Protocol A.2.
- **The protocol:**
  (1) The parties locally compute $[x] \cdot [y] - [r]_{2t}$ and send the result to party P_1 ($[x] \cdot [y]$ is locally computed by each party multiplying its own shares together).
  (2) Party P_1 use the first $2t$ shares it received from the parties and its own share to reconstruct $x \cdot y - r$, and then send it to all the other parties (Note that this is not the same as in the reconstruct procedure, as here there is no correctness check of the shares!).
  (3) The parties locally compute $[r]_t + (xy - r)$. Each party sets its result to be its share of $x \cdot y$.