Efficiently Compiling Secure Computation Protocols From Passive to Active Security: Beyond Arithmetic Circuits

Marina Blanton
University at Buffalo
Buffalo, NY, USA
mblanton@buffalo.edu

Dennis Murphy
University at Buffalo
Buffalo, NY, USA
dpm29@buffalo.edu

Chen Yuan*
Meta Platform, Inc.
Bellevue, WA, USA
chenyc@meta.com

ABSTRACT

This work studies compilation of honest-majority semi-honest secure multi-party protocols secure up to additive attacks to maliciously secure computation with abort. Prior work concentrated on arithmetic circuits composed of addition and multiplication gates, while many practical protocols rely on additional types of elementary operations or gates to achieve good performance. In this work we revisit the notion of security up to additive attacks in the presence of additional gates such as random element generation and opening. This requires re-evaluation of functions that can be securely evaluated, extending the notion of protocols secure up to additive attacks, and re-visiting the notion of delayed verification that points to weaknesses in its prior use and designing a mitigation strategy. We transform the computation using dual execution to achieve security in the malicious model with abort and experimentally evaluate the difference in performance of semi-honest and malicious protocols to demonstrate the low cost.

KEYWORDS

secure multi-party computation, secret sharing, active security, extended set of gates

1 INTRODUCTION

Secure multi-party computation techniques have experienced significant performance improvements in recent years and are now suitable for performing complex computation on large data sets. Historically, there has been a significant performance gap between techniques designed for the semi-honest (or passive) setting and the stronger malicious (or active) setting. A recent line of work [9, 21, 34] capitalizes on the notion of semi-honest protocols for arithmetic circuits *secure up to additive attacks* in the malicious model. Informally, security up to additive attacks means that corrupt participants are able to tamper with the computation of a gate only by adding some value to it. This property was utilized to build efficient protocols for arithmetic circuits in the honest majority setting secure against malicious adversaries with abort from semi-honest building blocks secure up to additive attacks. The result is efficient protocols in the malicious model (with abort) only slower by a small factor compared to the semi-honest version. Security with abort permits detecting misbehavior and terminating the computation, but not recovering from it. Fairness and guaranteed output delivery were also consequently considered [27, 35].

To this date, this line of work considered arithmetic circuits based on linear secret sharing. However, in practice many efficient protocols for implementing common operations such as comparisons in this setting are not arithmetic circuits consisting of only addition and multiplication gates. Instead, they rely on an extended set of elementary building blocks which, in addition to addition and multiplication gates, often uses simple operations such as opening a secret-shared value and generating a secret-shared random element. Let us use less-than comparisons as an illustrative example. All implementations of this operation based on secret sharing we are aware of in different tools and compilers (e.g., Sharemind [6], PICCO [40], SPDZ [18], and SCALE-MAMBA [36, 37]) are based on an extended set of elementary operations. Furthermore, we are not aware of a way to build an arithmetic circuit for this operation unless the operands are secret shared one bit at a time or a way perform bit decomposition to bit-decompose the inputs to comparison operations. This leaves us with running the entire computation on shares of individual bits if the computation includes comparisons, which can present significantly larger performance overhead compared to other known techniques.

As a simple example consider proximity testing which determines whether a squared Euclidean distance between two two-dimensional points (x_1, y_1) and (x_2, y_2) is above a certain threshold, i.e., $(x_1 - x_2)^2 + (y_1 - y_2)^2 > t$. When using an extended set of elementary operations for the comparison, the above function can be evaluated using 3ℓ interactive operations in 6 rounds, where ℓ is the bitlength of the operands in the comparison [8]. However, if we are constrained to the use of addition and multiplication gates and execute the entire computation on bit-decomposed values, the cost of multiplications becomes quadratic in the bitlength of the operands and the computation associated with additions and subtractions is no longer local. Furthermore, constant-round protocols for realizing many operations (such as comparisons, bit-wise additions or subtractions, etc.) also require tools beyond additions and multiplications. Thus, by constraining a protocol to arithmetic gates only we are paying a substantially higher price in terms of both the number of interactive operations and the number of rounds.

Summary of results. In this work we introduce new fundamental building blocks to the framework of Genkin et al. [21] that studied arithmetic circuits composed of addition and multiplication gates over a finite field $\mathbb{F}$ in the honest majority setting. In particular, we consider five additional protocols and use notation $[x]$ to denote that x is secret-shared among the participants:

*Majority of the work was performed while at the University at Buffalo.

Proceedings on Privacy Enhancing Technologies 2024(1), 74–97

https://doi.org/10.56553/popets-2024-0006

- $x \leftarrow \mathsf{Open}([x])$ reconstructs the value of its argument to the participants;
- $[x] \leftarrow \mathsf{RandFld}()$ generates a secret-shared (pseudo) random field element;
- $[x] \leftarrow \mathsf{RandInt}(k)$ generates a secret-shared (pseudo) random unsigned integer of at least k bits long;
- $z \leftarrow \mathsf{MulPub}([x], [y])$ reconstructs the value of $x \cdot y$;
- $[z] \leftarrow \mathsf{DotProd}(\langle [x_1], \ldots, [x_\ell] \rangle, \langle [y_1], \ldots, [y_\ell] \rangle)$ computes dot product $[z] = \sum_{i=1}^{\ell} [x_i] \cdot [y_i]$.

If we want to use this extended set of gates, first notice that it is no longer possible to securely evaluate arbitrary functions f. This is because two of the elementary gates (namely, Open and MulPub) disclose intermediate results of the computation. Therefore, our first step is to determine what functions can be securely evaluated in this framework by placing constraints on the use of these special gates. We are not aware of this types of analysis to be conducted in prior literature for general functions and provide conditions for when input-dependent values are properly protected to be disclosed as part of secure evaluation. In particular, input-dependent values can be protected by means of perfect or statistical hiding and the computation can also reconstruct values the computation of which did not involve inputs (e.g., randomly generated elements).

The next step is to show our new building blocks secure up to additive attacks if we would like to be able to apply the logic of efficient compilers developed for arithmetic circuits with this property. The framework of [21] permits only the entire computation, consisting of all necessary protocols, to be shown secure up to additive attacks and not a specific protocol on its own. We thus extend the framework of [21] to cover additional types of gates. In doing so, we need to extend the definition of linear-based protocols and prove its instantiation weakly secure, which according to the logic in [21] results in security up to additive attacks. A crucial component of this framework is that the underlying secret sharing scheme is dense and redundant. This permits extraction of a secret-shared value from the shares of honest parties in the presence of honest majority and a consistent view of the honest parties is necessary for showing security. However, this property no longer holds for semi-honest protocols with share reconstruction such as Open where an adversary can force the honest parties to reconstruct different values and diverge their views. Replacing building blocks that perform reconstruction with maliciously secure variants is also not permissible because the computation is restricted to a linear combination of input-dependent messages. As a result, we extend the set of building blocks secure up to additive attacks with gates that do not perform reconstruction, namely, randomization gates and the dot product. The gates that perform reconstruction are later directly instantiated with maliciously secure variants which have efficient realizations.

The high-level structure of the conversion from semi-honest building blocks secure up to additive attacks to maliciously secure computation (for large fields) follows the idea of dual circuit execution on the original and randomized inputs and checking them for consistency as used in [9] and earlier work. That is, the parties generate a random field element $[r]$, execute the function on the inputs as before and in parallel execute the function on randomized values, where for each wire with value $[x]$ in the first circuit, the second circuit contains value $[r \cdot x]$. Then prior to opening the results of the computation, the parties check the output of each tamperable gate in the two executions for consistency and abort if a difference is detected. We note that more recent results such as [26, 27] permit verification of multiplication gates without having to execute the computation the second time and thus result in a more efficient solution. We anticipate that those techniques might be applicable to our setting as well and leave it as a direction of future work. Note that the only gates that require verification in our instantiation are multiplication and dot product gates.

The introduction of reconstruction gates such as Open presents challenges, one of which is that we cannot disclose the reconstructed value x together with its randomized version $r \cdot x$ (which would disclose r) and thus we only reconstruct x (which typically re-enters the computation as a constant). More fundamentally, the presence of gates that generate randomness such as RandFld and RandInt makes it possible to reconstruct a secret-shared value (via a call to Open) generated according a distribution that spans a subset of the field $\mathbb{F}$ instead of the entire field. For example, the computation might generate a random bit (which is typically done by squaring a random element and using its smaller square root to determine the outcome with 50% probability), XOR it with an input-dependent bit b to achieve perfect hiding, and reconstruct the result. While this type of computation complies with the security requirements in the semi-honest model, it becomes problematic for any framework that uses delayed verification. That is, if the bit b is the result of prior computation which is subject to adversarial tampering, its value may not correspond to a bit and XOR will no longer protect the value from disclosure. The implication is that in the presence of an extended set of gates, it is no longer safe to universally delay verification to the end of the computation.

Our solution is to determine a conservative condition when it would be safe to proceed with Open without verifying correct execution of all prior gates and mark all Open gates in a circuit not compiling with this condition as having to trigger verification prior to performing the reconstruction. Our condition requires protecting input-dependent values with a correctly generated (i.e., not tamperable) random field element, where the protected value is also distributed over the entire field (i.e., addition can be used, but not multiplication). We then design an algorithm for traversing the circuit and using gate and function information to mark each Open gate as having to trigger verification prior to the opening or not.

The requirements above inform our transformation from building blocks secure up to additive attacks to an execution secure in the malicious model. We start with the structure from [9] that uses a hybrid model, introduce additional ideal functionalities corresponding to computation of different gates, and provide the compiler itself. We show security in the large field setting and describe how to adopt the construction to a smaller field.

We implement the compiler and present experimental results that show the runtime of semi-honest functionalities and the corresponding maliciously secure variants produced using our framework. Our empirical evaluation shows that maliciously secure functionalities are slower by a small multiplicative factor, typically 3–4. We also show that the presence of the extended set of gates can improve performance by up to 3 orders of magnitude for some functionalities.

Our compiler is applicable to any framework that implements secure multi-party protocols over a finite field in the semi-honest setting with honest majority and uses the extended set of elementary operations outlined above. This includes protocols based on the work of Catrina and de Hoogh [8] and follow-up work, which were implemented in PICCO [40], in SCALE-MAMBA [37], and were adopted to the dishonest majority setting in SPDZ implementation. Furthermore, certain components of this work, namely, the conditions for when function f can be securely evaluated in the presence of the extended set of elementary gates, are also applicable to both honest majority and dishonest majority settings. Lastly, our next immediate goal is to extend the techniques to computation over a ring $\mathbb{Z}_{2^k}$ as well.

To summarize, our contributions are as follows:

- We extend the framework of Genkin et al. [21] of protocols secure up to additive attacks composed of two types of arithmetic gates to support fundamentally different types of gates that permit randomization and opening of protected intermediate results. This involves generalization of the notion of linear protocols, linear-based protocols, and providing corresponding rigorous security proofs.
- The presence of open gates has a profound impact on secure function evaluation of circuits and it is no longer safe to evaluate arbitrary circuits. Thus, we formulate conditions on the circuit structure under which it is safe to perform secure function evaluation.
- To use the extended framework of protocols secure up to additive attacks, we instantiate all gates with specific protocols and prove that the resulting construction satisfies the necessary properties.
- To transform protocols secure up to additive attacks to protocols secure in the malicious model, the notion of delayed verification is often used. In this work we show that it is not always safe to delay verification of extended circuits until the end of the computation (pointing to weakness in prior implementations), formulate conservative conditions for when it is safe to do so, and present an algorithm that analyzes a circuit and triggers verification according to the circuit structure to preserve security.
- Lastly, we combine all of the above results in a compiler that transforms a protocol secure up to additive attacks composed of the extended set of gates to a protocol secure in the malicious model and prove its security. Our compiler is an extension of the transformation used in [9].
- We implement the protocols for sample functionalities before and after the transformation and show the the cost of achieving active security from passively secure protocols is low. We also draw a comparison with alternative solutions.

Related work. Genkin et al. [21] extend a previous work by Cramer et al. [12] to protect computation within arithmetic circuits against attacks by malicious parties which are equivalent to the addition of a value, predetermined by an adversary, to any wire within the circuit during protocol execution. This general construction was realized with efficient batched verification in work by Lindell and Nof [34] along with performance evaluations, and improved upon by Chida et al. [9]. Additional newer results include [7, 25–27] that further lower the cost of verification and achieve low communication complexity, on par to that in the semi-honest model. All of them work with arithmetic circuits consisting of two types of gates. Furukawa and Lindell [20] develop a solution that assumes a 2/3 honest majority. Genkin et al. [23] developed a framework to extend [21] into the realm of boolean circuits. Hazay et al. [29] proved the BMR garbling construction in [4] to be secure up to additive attack, and observed that the parallel functionality can be optimized in certain cases to be of lower overhead than the original circuit. They built these observations into a compiler for boolean circuits which are secure in the 1-bit paradigm. Hazay et al. produced another work [30] which uses oblivious transfer (OT) or oblivious linear function evaluation (OLE) in a compiler that works for boolean or arithmetic circuits, respectively.

Another line of research was initiated by SPDZ [18]. In this setting, secret shared values are protected via shared MACs. This framework is aided by somewhat homomorphic encryption and preprocessing, and is able to tolerate dishonest majority. The work was consequently extended to support computation over ring $\mathbb{Z}_{2^k}$ [10, 14]. That setting is also relevant to our context: while statistical hiding (and invocation of RandInt gate) is normally not used with computation modulo 2^k, an unconstrained combination of other gates can still lead to information disclosure, as we demonstrate in section 5.1. To that extent, our next immediate goal is to apply the results of this work to the setting of rings $\mathbb{Z}_{2^k}$. That is, Abspoel et al. [1] is a follow-up work on Chida et al.'s compiler that builds underlying tools for the transformation to support computation over rings. While that work still only supports pure arithmetic circuits, we can use it as the basis for expanding supported circuits with additional types of gates.

Escudero et al. [19] provide efficient generation of shares of random bits over $\mathbb{Z}_2$ and their corresponding integer value in a larger field or ring for use in mixed boolean-arithmetic circuits. The technique, edaBits, instructs a party to enter related inputs into the computation and correctness in the malicious setting is achieved using a new variant of a cut-and-choose technique. The techniques reduce the cost of random bit generation and use in non-linear operations, but does not treat the topic of delayed verification in the presence of calls to open as we do. We comment on relative performance of the techniques in section 6. Lastly, Dalskov et al. [13] provide a dot product protocol in the malicious model similar to the treatment of our dot product gate; however, we were unable to find its (security) analysis. Our similarity to Chida et al. [9] (on which we build) is limited to section 5.2, where our Construction 2 extends their transformation with four new types of gates, has a modified verification phase, and requires separate security analysis.

2 SETUP

Secret sharing is a fundamental technique that allows a dealer to produce n shares of a secret so that any $\leq t$ shares reveal no information about the secret, while any subset that contains $t+1$ or more shares allows for efficient reconstruction of the secret. We refer to t as the threshold and in this work we assume the setting with honest majority, namely, that $t < n/2$. Of particular importance for secure multi-party computation are linear secret-sharing schemes, which enable local computation of a linear combination of secret

shared values. Throughout this work we assume that secret sharing is set up over a finite field $\mathbb{F}$.

We use the notation $[x]$ to represent a secret-shared x using a linear secret sharing scheme, where $[x]_i$ corresponds to the share of x held by party i. We require that the (n, t) secret sharing scheme we employ over $\mathbb{F}$ supports the following functions:

- share(x): On input private $x \in \mathbb{F}$, this function generates n shares $\{[x]_1, [x]_2, \ldots, [x]_n\}$, where $[x]_i$ denotes the share intended for party P_i.
- share(x, $[x]_J$): Given a subset J of the parties of size $|J| \leq t$, this function takes as input $|J|$ shares, $[x]_J \in \mathbb{F}^{|J|}$ and private $x \in \mathbb{F}$ and generates the remaining shares according to the input. Note that when $|J| = t$, $[x]_J$ uniquely determines the output shares.
- reconstruct($[x]_J$): Given exactly $|J| = t+1$ shares $[x]_J$, this function reconstructs the secret and outputs x.

The functions above are algorithms that specify how to create shares of a secret and reconstruct a secret from its shares. They are not intended to be resilient to malicious behavior and additional mechanisms are needed to strengthen them for achieving equivalent functionalities in the presence of malicious players.

Our main construction in Section 5 that transforms computation with building blocks secure up to additive attacks to secure computation with abort in the presence of malicious adversaries can be instantiated using any suitable secret sharing scheme over finite field $\mathbb{F}$ such as Shamir secret sharing [39] or replicated secret sharing [32], as long as the expectations for the sub-functionalities are met. The instantiation of the sub-functionalities that we provide in this work in Section 4 is based on Shamir secret sharing (and some components of the protocols make use of replicated secret sharing for performance reasons). Therefore, notation $[x]$ refers to the main secret sharing scheme, which in Section 4 is instantiated with Shamir secret sharing.

To permit non-interactive share generation, e.g., generation of a random field element by RandFld, we utilize replicated secret sharing (RSS) [32] with the same access structure and threshold t. It is an additive secret sharing scheme with each party holding multiple shares and we denote x secret-shared using RSS by $[\![x]\!]$. Our use of RSS is limited to the following high-level idea: the parties hold $[\![\text{key}]\!]$, use shares of key as seeds into a pseudo-random generator (PRG) to non-interactively obtain shares of a pseudo-random element, and locally convert their computed shares into Shamir shares of the desired value. This approach is used to realize gates RandFld, RandInt(k), and generation of fresh Shamir shares of 0 with threshold $2t$. We consequently denote Shamir shares of x with threshold $2t$ by $\langle x \rangle$. With (n, t) Shamir secret sharing, a secret x is represented by a random polynomial of degree t with the free coefficient set to x. Each party's share corresponds to evaluation of the polynomial on a unique non-zero point and, given $t + 1$ or more shares, a secret is reconstructed via Lagrange interpolation.

Let Γ be an access structure which in our context permits access for any subset of $t + 1$ or more parties and let $\mathcal{T}$ be the union of all maximal unqualified set of Γ, i.e., subsets of parties of size t. Then party P_i holds RSS shares $x_T \in [\![x]\!]_i$ for each $T \in \mathcal{T}$ such that $P_i \notin T$. At the setup time, the parties generate $[\![\text{key}]\!]$. We use PRG $\mathbb{F}$PRG that outputs field elements and PRG $\mathbb{Z}$PRG$_k$ that outputs k-bit integers. When we make a call to a PRG in a protocol, it should be understood that it returns the next element in the sequence. Notation $[a, b]$ defines the range $a, \ldots, b$.

3 FUNCTION REQUIREMENTS

Before we can present our compiler from building blocks secure up to additive attacks to function evaluation secure in the presence of malicious participants with abort, we must determine what constraints should be placed on functions we are evaluating. The rationale is that in the presence of Open (and MulPub) gates a function can disclose private data during the computation and we would like to eliminate from consideration functions that leak information.

Opening (randomly generated or properly protected) intermediate results is common in protocols for specific operations and does not compromise security guarantees. For example, the use of Open appeared in 1989 for secure computation of the inverse of a field element and unbounded fan-in multiplication [2], is widely used in multiplication protocols based on triple generation starting from [3], is common in more complex protocols, e.g., in [8, 15] and others. MulPub was introduced as an optimization that combines multiplication followed by Open (used in [8] and after) and is extensively used in many protocols such as random bit generation, all types of comparisons, etc. While each individual protocol among these examples that reconstructs an intermediate result (by means of Open or MulPub) can be shown to be secure, we are not aware of any general analysis of requirements which function f must satisfy in the presence of Open gates to be able to maintain the expected level of security. Our analysis led us to formulating the requirements that a function f that uses share reconstruction prior to the output recovery must satisfy to meet the requirement of no information leakage in the way specified below. Throughout this work, we let M denote the number of inputs into function f, each represented as an element of field $\mathbb{F}$, and use notation v_i to denote the ith input. Each party contributing input into the computation can supply multiple elements v_i.

Definition 3.1. Function f being evaluated must satisfy the requirements that for any variable opened during the computation one of the following conditions hold:

(1) the variable has a known distribution over a known subset $B \subseteq \mathbb{F}$ independent of all inputs $v_1, \ldots, v_M$, i.e., perfect secrecy is achieved, or
(2) the distribution of the variable is statistically close to the distribution of a random variable with a known distribution over known subset $B \subset \mathbb{F}$ independent of all inputs $v_1, \ldots, v_M$, i.e., statistical secrecy is achieved, or
(3) the variable was not computed as a function of any of the inputs $v_1, \ldots, v_M$ and has an arbitrary distribution over a known subset $B \subseteq \mathbb{F}$.

This definition states that if a value was produced without using any of the private inputs or any information derived from the inputs (condition 3), disclosing the value does not reveal any sensitive information and therefore is safe. It is also safe to disclose a value if it was computed using one or more inputs or information derived from them, but input-dependent information is perfectly protected

(condition 1) or statistically protected (condition 2). Computationally secure protection mechanisms are not useful as elementary building blocks for protocol design and do not apply when the field size is not sufficiently large (which is the case for information-theoretic techniques).

This definition suggests that if a value is generated without using any inputs or intermediate results that depend on the inputs, it can be from any subset of $\mathbb{F}$ and any distribution we can sample from. For instance, the computation can generate a random bit and consequently open it. If the value being opened was computed as a function of the inputs, there are two common ways to protect a private intermediate result in $\mathbb{F}$: by multiplication or by addition with a random element. In particular, if r is an element drawn uniformly at random from $\mathbb{F}$ and a is input dependent, it is safe to open $r + a$ (i.e., both result in uniform distributions). We could also achieve statistical security by opening $r + a$ if a is an input-dependent value of bitlength k and r is a value drawn uniformly at random from the set of integers of bitlength $k + \kappa$ (or larger space), with κ being a statistical security parameter. In addition, if a is input-dependent *non-zero* element of the field and r is drawn uniformly at random from $\mathbb{F}$ (or $\mathbb{F} \setminus \{0\}$), it is safe to open $r \cdot a$.

The above implies that, if f uses any MulPub gates, either both inputs to that gate must be computed without using f's inputs (and in that case MulPub's inputs can be sampled from any space) or one of the MulPub inputs is not a function of f's inputs and is sampled uniformly at random from the entire field and the other MulPub's input is guaranteed to be non-zero. Also, if a randomly sampled value was opened once as part of f, it cannot be used for protection of input-dependent variables again because input-independent distribution is not achievable.

We would like to use sample functionalities RandBit and Trunc to illustrate how the constraints of definition 3.1 are satisfied in typical protocol designs. The goal of RandBit is to let the participants generate a private random bit on no input. Its realization from [8] starts by generating a private random field element $c \in \mathbb{Z}_p$ (where p is prime such that $p \bmod 4 = 3$), squaring it, and reconstructing the square. This behavior falls into the third category of definition 3.1: the disclosed value is not a function of private inputs and subset B consists of quadratic residues modulo q and 0.

Another example is truncation Trunc and various forms of comparisons, which start by generating a private random integer at least κ bits longer than their private input, where κ is a statistical security parameter, adding the input to the generated integer, and disclosing the sum to the participants. This falls in category 2 of definition 3.1, where once again B is known and determined by the algorithm and the distribution is uniform.

To prove evaluation of functions specified in Definition 3.1 secure, we would need that for each opened variable distributed over some $B \subseteq \mathbb{F}$, an element of B is efficiently (probabilistic polynomial time in a security parameter) sampleable according to the distribution used in f. This is straightforward to do for all cases of Definition 3.1 because sampling for all (PPT) functions f can be achieved by simply performing the computation the way f does (recall that the distributions are required to be input independent).

4 TOWARD FUNCTION EVALUATION SECURE UP TO ADDITIVE ATTACKS

To be able to build our compiler, we first need to demonstrate security up to additive attacks of secure function evaluation using our building blocks. The framework of Genkin et al. [21, 22] states that if a linear-based protocol (as defined in [22]) is weakly private against active adversary $\mathcal{A}$ controlling at most t computational parties, it is actively secure up to additive attacks. This means that showing security up to additive attacks is performed by showing that a linear-based protocol is weakly private.

4.1 Extended Linear-Based Protocols

Upon closer examination, we determined that prior to using this logic to demonstrate security up to additive attacks we need to revisit the notion of linear-based protocols. In particular, a linear-based protocol was defined in [22] for a redundant dense linear secret sharing scheme (e.g., Shamir secret sharing) characterized by share and reconstruct functions as consisting of a setup, randomness generation, input sharing, circuit evaluation, and output recovery phases. The important component is that each gate computed as part of circuit evaluation must correspond to a linear protocol. Informally, each participant in a linear protocol is permitted to locally compute any function of its inputs and send messages to other participants, but once messages from other participants are received, the output can only be computed as a linear combination of the incoming messages [22].

Because we are expanding the set of gates that secure function evaluation executes, in our context it is necessary to expand the definition of a linear-based protocol with new types of gates, namely opening, randomness generation, opened multiplication, and dot product. In doing so, we first re-structure the definition of a linear protocol and consequently use it in defining a linear-based protocol. Specifically, circuit evaluation in Genkin et al.'s framework involved only one type of non-trivial gate (i.e., multiplication) and any input-independent pre-computation associated with evaluation of that gate (e.g., triple or randomness generation) was incorporated into the setup phase. In our context, different gates may rely on different types of pre-computation and, instead of executing each pre-computation a necessary number of times in the setup phase, we permit the gate evaluation itself to be split into input-independent pre-computation and the main computation, while preserving the same restrictions as before. This allows for added flexibility without changing the essence of the definition. We obtain the following definition of a linear protocol:

Definition 4.1 (Linear protocol). An n-party protocol Π is a linear protocol over some finite field $\mathbb{F}$ if it is specified as follows:

(1) ***Input.*** The input of every party P_i consists of values distributed at the setup phase (of arbitrary type) and inputs into the protocols specified as a vector of elements from $\mathbb{F}$. The former are called setup-based inputs and the latter are main inputs.
(2) ***Pre-computation.*** The parties engage in a linear protocol π_{precomp} using setup-based inputs as the main inputs (and no other inputs). The result consists of each party obtaining a vector of field elements, which are called auxiliary inputs.

(3) ***Messages.*** Every message in Π is a vector of field elements. A message $\vec{m}$ sent by some party P_i (possibly to itself) belongs to one of the two categories:
   (a) $\vec{m}$ is some fixed arbitrary function of P_i's main inputs (and is independent of its auxiliary inputs). We refer to this as a Type A message.
   (b) every element of $\vec{m}$ is generated as some fixed linear combination of P_i's auxiliary inputs and elements of previous messages received by P_i (including of Type A above). We refer to this as a Type B message.

(4) ***Output.*** The output of every party P_i is a linear function of its incoming messages.

We can now formulate an extended linear-based protocol. Following prior work, it is formulated as n-party computation with the inputs provided by m clients (i.e., the inputs can come from participants other than the computational parties) and a single output.

Using the current framework, we were unable to prove a semi-honest version of Open to be secure up to additive attacks because the honest parties are not guaranteed to possess the same opened value at the end of Open due to the adversary's ability to communicate inconsistent values to other parties. On the other hand, a malicious version of Open where honest parties perform cross-checking to validate the result would not comply with with the definition of a linear protocol above. For that reason, we exclude gates that perform reconstruction from shares from our formulation of extended linear-based protocol and will directly utilize a maliciously secure realization of Open in our final solution.

Throughout this work, we formulate RandInt gate as, on input k, drawing an integer from the range 0–$c2^k$, where $c \geq 1$ is a fixed constant. This formulation preserves the necessary security properties while supporting efficient realizations of this gate.

Definition 4.2 (Extended linear-based protocol). Let $\mathcal{SS}$ = (share, recover) be a redundant dense linear secret sharing scheme. An n-party m-client protocol Π for computing a single-output function $f : \mathbb{F}^{I_1} \times \cdots \times \mathbb{F}^{I_m} \to \mathbb{F}^{O_1}$ with $n = 2t + 1$ is linear-based with respect to $\mathcal{SS}$ if Π has the following structure:

(1) ***Setup phase.*** The computational parties perform π_{setup}, a one-time setup protocol (such as key distribution) based on the properties of the computation.

(2) ***Input sharing phase.*** P_i shares its input v_i using the share() functionality of $\mathcal{SS}$.

(3) ***Circuit evaluation phase.*** Π computes the circuit by evaluating each gate in some topological order. After honest execution of gate i that produces shares, the parties hold shares of the gate's output a distribution induced by share. The evaluation of each gate is performed as follows:
   (a) For any addition (or subtraction) gate i with inputs from gates j and k, Π evaluates gate i by having each party add (resp., subtract) its shares corresponding to the outputs of gates j and k.
   (b) For any gate i that performs multiplication of one input from gate j and a constant field element, Π evaluates gate i by having each party multiply its share of gate j's output by the specified constant.
   (c) For any multiplication gate i with inputs from gates j and k, Π evaluates gate i using some n-party linear protocol π_{mult}, where each party uses its shares of gates j and k's outputs as the inputs to π_{mult}.
   (d) For any dot product gate i with inputs from gates $j_1, \ldots, j_\ell$ and $k_1, \ldots, k_\ell$, Π evaluates gate i using some n-party linear protocol π_{dotprod}, where each party uses its shares corresponding to the outputs of gates $j_1, \ldots, j_\ell$ and $k_1, \ldots, k_\ell$ as the inputs to π_{dotprod}.
   (e) For any random field generation gate i, Π evaluates gate i using some n-party linear protocol π_{randfld} that does not take inputs from any other gate.
   (f) For any random integer generation gate i, Π evaluates gate i using some n-party linear protocol π_{randint} that does not take inputs from any other gate.

(4) ***Output recovery phase.*** During output recovery of Π, the first $t + 1$ parties send their shares of each output gate to the output recipient, who consequently recovers each output using recover.

Given a specification of a linear-based protocol, security of a specific instantiation of that protocol interface up to additive attacks is shown in two steps, using the notion of so-called weakly-private protocols. In particular, the notion of weak privacy considers a truncated protocol view, without the last communication round in which outputs are disclosed, and is defined below. Then a specific instance of a protocol for circuit evaluation first needs to be shown to be linear-based and weakly-private. Second, it needs to be shown that a protocol which is both weakly-private and linear-based is a protocol secure up to additive attacks in the presence of active adversaries. The latter was shown in [22] for a formulation of a linear-based protocol evaluating an arithmetic circuit consisting of two types of gates, while we prove the same property for our formulation of an extended linear-based protocol evaluating a circuit consisting of an extended set of gates in Appendix E. Thus, the next step will be to instantiate all components of our extended linear-based protocol with concrete sub-protocols and show that the resulting construction is linear-based and achieves weak privacy.

4.2 Weakly-Private Protocols

We start by defining the notion of weakly-private protocols:

Definition 4.3 (Weakly-private protocol; adopted from [22]). Let π be an n-party protocol for computing functionality $f : \mathbb{F}^{I_1} \times \cdots \times \mathbb{F}^{I_m} \to \mathbb{F}^{O_1}$ and let $\mathcal{A}$ be an adversary controlling at most t computational parties. Also let $\mathsf{view}_{\mathcal{A}}^{\pi,\mathsf{trunc}}(\vec{x})$ denote the view of adversary $\mathcal{A}$ excluding the last communication round during a real execution of protocol π on inputs $\vec{x}$. We say that π is weakly-private against $\mathcal{A}$ if there exists a simulator $\mathcal{S}$ such that for every input $\vec{x}$ the real and simulated views are indistinguishable, i.e., $\mathsf{view}_{\mathcal{A}}^{\pi,\mathsf{trunc}}(\vec{x}) \equiv \mathcal{S}(\vec{x}_C)$.

As before, the function f being evaluated needs to comply with the requirements of Definition 3.1. The computation and the corresponding protocols are specified as Construction 1.

Recall that the computation in RandFld and RandInt uses RSS with a previously set up secret $[\![\mathsf{key}]\!]$ and consecutive share conversion to Shamir secret sharing. For a secret-shared $[\![x]\!]$, let notation x_Q represent a share given to all parties in Q, i.e., $P_i \in Q = [1, n] \setminus T$ for each $T \in \mathcal{T}$, and let $\mathcal{Q}$ be the set of all Qs, i.e., $\mathcal{Q}$ is all subsets

of $[1, n]$ of size $n - t$. RandFld and RandInt from [11, 17] then use unique polynomials λ_Q of degree t for each Q such that $\lambda_Q(0) = 1$ and $\lambda_Q(i) = 0$ for all $P_i \notin Q$.

We present two multiplication protocols: one is based on GRR multiplication [24] with changes from [5] to reduce communication and the second is based on DN multiplication [16] with communication reduction from [5]. We refer to the former as a single-round and to the latter as a linear-communication protocol. The former is beneficial for the common case of three parties (lower communication and rounds), while the latter lowers communication when the number of computational parties grows and uses two rounds. As a result, we obtain low communication of 1 field element per party in the three-party setting (single round) and $\lesssim 2$ field elements per party on average with larger n (two rounds). This communication matches the state of the art semi-honest communication of multiplication in [25] which continues the line of work on compiling semi-honest arithmetic circuits into maliciously secure protocols.

The linear-communication multiplication relies on non-interactive generation of pseudo-random sharing of zero (PRZS) $\langle 0 \rangle$ from [11]. PRZS computation is input-independent. Because arbitrary computation is permitted in the pre-computation phase of a linear protocol, the details of PRZS are not essential and can be found in [11]. We denote this protocol by PRZS. DN multiplication [16] dedicates one party as the king and uses a fixed (constant) representation of a sharing of 1, [1].

The single-round multiplication uses interpolation constants α_i for party P_i as defined in [24]. In addition, party P_i shares a key for pseudo-random number generation with each of other t parties. For simplicity and without loss of generality, we use a symmetric setup in which P_i shares individual keys $k_{i,j}$ with parties P_j for $j = (i + 1) \bmod n, \ldots, (i + t + 1) \bmod n$. This establishes PRG keys between each pair of parties P_i and P_j.

Construction 1. *Building Blocks Secure Up to Additive Attacks.*

(1) Input sharing: Each input owner calls $\mathsf{share}(x)$ *on its input* x *and sends share* $[x]_i$ *to party* P_i.

(2) Function evaluation: Evaluate each gate in the given topological order as follows:

(a) RandFld *gate: Each party* P_i, $i \in [1, n]$, *calculates:*

$$[x]_i = \sum\nolimits_{Q \in \mathcal{Q}, i \in Q} \mathbb{F}\mathsf{PRG}(\mathsf{key}_Q) \cdot \lambda_Q(i)$$

(b) RandInt *gate: On input* k, *each* P_i *calculates:*

$$[x]_i = \sum\nolimits_{Q \in \mathcal{Q}, i \in Q} \mathbb{Z}\mathsf{PRG}_k(\mathsf{key}_Q) \cdot \lambda_Q(i)$$

(c) Addition gate: On input $[x]$ *and* $[y]$, *each* P_i *calculates* $[z]_i = [x]_i + [y]_i$.

(d) Multiplication by a known field element: On input $[x]$ *and* $y \in \mathbb{F}$, *each* P_i *calculates* $[z]_i = [x]_i \cdot y$.

(e) Multiplication gate (linear comm.): Pre-computation: Execute $\langle 0 \rangle \leftarrow \mathsf{PRZS}()$ *and* $[w] \leftarrow \mathsf{RandFld}()$ *and each* P_i *computes* $\langle u \rangle_i = [w]_i \cdot [1]_i + \langle 0 \rangle_i$.
Input-dependent computation:
(i) On input $[x]$ *and* $[y]$, *each* P_i *computes* $\langle z \rangle_i = [x]_i \cdot [y]_i$, $\langle v \rangle_i = \langle z \rangle_i + \langle u \rangle_i$ *and sends* $\langle v \rangle_i$ *to the king.*
(ii) The king reconstructs v *from* $2t+1$ *shares* $\langle v_i \rangle$ *and sends it to each party.*
(iii) Each P_i *computes* $[z]_i = v - [w]_i$.

(f) Multiplication gate (single round):
Pre-computation: Each P_i *sets* $[d_i]_j = \mathbb{F}\mathsf{PRG}(k_{j,i})$ *for* $j = 1 + (i \bmod n), \ldots, 1 + ((i + t) \bmod n)$.
Input-dependent computation:
(i) On input $[x]$ *and* $[y]$, *each* P_i *computes* $\langle z \rangle_i = [x]_i \cdot [y]_i$ *and sets* $[d_i]_0 = \langle z \rangle_i$.
(ii) Each P_i *reconstructs polynomial* $f_{\langle z \rangle_i}$ *from* $t + 1$ $[d_i]_j$*s.*
(iii) Each P_i *evaluates and sends* $[d_i]_j = f_{\langle z \rangle_i}(j)$ *to* P_j *for* $j = 1 + ((i + t + 1) \bmod n), \ldots, 1 + ((i + n - 2) \bmod n)$.
(iv) Each P_i *computes* $[z]_i = \sum_{j=1}^{n} \alpha_j [d_j]_i$.

(g) Dot product gate: The computation is the same as in one of the multiplication gates above except that the inputs are $([x_1], [x_2], \ldots, [x_\ell])$ *and* $([y_1], [y_2], \ldots, [y_\ell])$ *and each* P_i *computes* $\langle z \rangle_i$ *as* $\langle z \rangle_i = \sum_{j=1}^{\ell} [x_j]_i \cdot [y_j]_i$.

Here, RandInt from [11] produces an integer from the range 0–$v2^k$, where v is the number of shares. As discussed before, this is suitable for its use in protocols with statistical hiding. This is reflected in our analysis and the corresponding ideal functionality. All of these building blocks either follow from the properties of the secret sharing scheme (i.e., addition and multiplication by a known field element) or were presented and analyzed in prior work (i.e., RandFld and RandInt from [11], multiplication protocols from [5], and the dot product follows from the properties of the secret sharing scheme and multiplication protocols). For that reason, their correctness and security (in the semi-honest model) follow from prior work, but for completeness we examine correctness in more detail in Appendix A. These building blocks (or their older variants) have also been used in general-purpose (secret sharing based) secure multi-party computation compilers such as PICCO [40] and SCALE-MAMBA [37]. In this work, we take one step further: we prove stronger security properties which permit their use in an efficient transformation to a maliciously secure protocol.

To achieve our goal, we first need to show that Construction 1 achieves weak privacy, which we state as follows:

Theorem 4.4. *Construction 1 for evaluating a function* f *satisfying the constraints in Definition 3.1 is weakly-private in the presence of an active adversary* $\mathcal{A}$ *controlling at most* t *computational parties.*

The proof that constructs a simulator and shows indistinguishability is given in Appendix A. Throughout the rest of this work, we let C denote the set of corrupt parties, H be the set of honest parties, and $[x]_C$ and $[x]_H$ represent the union of shares held by the corrupt and honest parties, respectively.

To show the computation in Construction 1 secure up to additive attacks, what is remaining to demonstrate that the building blocks are linear protocols according to Definition 4.1. This, together with the logic described above (and information provided in Appendix E) will give us the following result:

Lemma 4.5. *Construction 1 is secure up to additive attacks.*

The proof which primarily shows compliance with Definition 4.1 is provided in Appendix A.

Before we continue, we would like to discuss the difference between Open protocols (not included in the specification of a linear-based protocol) and DN-style linear multiplication which internally reconstructs a secret-shared value, similar to the computation an Open protocol would. The difference comes from the

input-output interface of Open (and the requirement for it to be a linear protocol). In particular, showing that a construction is a linear-based protocol requires that each sub-protocol (such as multiplication) produces a value that a simulator can extract from the outputs of honest parties (see Appendix E for more detail). In the case of sub-protocols that generate a shared value, we discard outputs of malicious parties and can reconstruct a field element from the shares of the honest parties, which is always possible because the secret sharing scheme is dense and redundant. This also means that shares across all parties can be inconsistent, but this does not prevent the logic of the proof from proceeding. However, in the case of Open, a malicious participant can cause the honest parties to produce different outputs. In that case, it is not possible for the simulator to use a single value as the output of that sub-protocol. Furthermore, if we strengthen our realization of Open to perform verification and eliminate inconsistencies, the computation is no longer linear.

However, because a malicious adversary is permitted to deviate from the protocol in an arbitrary way, a malicious king can, for example, send $2v$ instead of v in the multiplication protocol. This would result in the parties computing $2v - w = 2a \cdot b + w$. Because additive attacks introduce errors independent of the wires, this would be treated as going beyond additive attacks. However, the fact that DN multiplication was shown to be secure up to additive attacks as part of a larger construction in [22] points out to an inconsistency in prior work and the need to treat Open operations with care.

5 MAIN CONSTRUCTION

Having developed the necessary building blocks secure up to additive attacks, we next build a compiler for producing execution resilient to active attacks. Note that we incorporate an Open gate directly instantiated with a maliciously secure protocol, which can be efficiently realized in the same way as in prior work, e.g., [9]. In particular, to reconstruct an element from shares, the parties communicate their own share to all other participants and reconstruct from all subsets of $t + 1$ shares checking for consistency. Adding a MulPub gate will also require a maliciously secure version, which unlike the semi-honest setting does not lead to efficiency improvements of combining multiplication and Open into a single gate. For that reason, we do not include an explicit MulPub gate.

Before we proceeding with the compiler, we need to analyze the impact of delayed verification on security in the presence of Open and randomization gates and show in what circumstances it is safe to delay verification and when opening a potentially tampered value can lead to information disclosure. In our attacks we conservatively assume that only multiplication (or dot product) gates can be tampered with. This is because instantiations of several building blocks in Construction 1 are non-interactive and therefore are resilient to malicious behavior. In particular, all of addition, multiplication by a known element, generation of a random field element and a random integer are non-interactive. An instantiation of Open will also have resilience to malicious behavior. This means that if delayed verification is not safe by tampering with only multiplication (or dot product) gates, a modification to the strategy of delaying verification until the end of the computation is necessary.

5.1 On Delayed Verification

Delayed verification is a common mechanism for achieving security in the presence of active adversaries including SPDZ [18] and its follow-up work with dishonest majority and more recent solutions [9, 34] that assume honest majority. These solutions treat computation expressed as an arithmetic circuit and delay verification of the computation until the end, i.e., right before disclosing the output. In this work we find that the ability to reconstruct values during function evaluation introduces a possibility to learn information about private inputs if verification is performed at the end of the computation, which might require changes to prior solutions. It is informative to consider reconstruction of different types of values.

SPDZ [18] (which is not constrained to additive attacks) uses calls to Open during evaluation of multiplication gates. In particular, to multiply $[x]$ and $[y]$, the parties utilize a pre-generated multiplication triple ($[a]$, $[b]$, $[c = ab]$) and open $x - a$ and $y - b$ (in $\mathbb{F}$). In this case, because a and b are random field elements and because the triple is verified to be well formed, the opened values information-theoretically protect x and y regardless of what values x and y might take.

However, if we look at other uses of Open operation in typical protocols, the situation changes. For example, as previously mentioned, truncation and comparison protocols use statistical hiding and execute computation of the form

$c \leftarrow \mathsf{Open}([x] + [r]);$

where r is a random integer at least κ bits longer than the value x it protects and, as before, κ is a statistical security parameter. For the sake of this example, let x be a k-bit integer when the computation is performed correctly, which means that r is at least $\kappa + k$ bits long. Now notice that x here can be a result of prior computation that includes multiplication operations susceptible to additive attacks. For example, if x was computed using two multiplications, e.g., as $x = abc$ using private a, b, and c, it is easy for the adversary to make x exceed k-bit length through additive attacks and lead to information disclosure about a private value. That is, after multiplication $a \cdot b$ we can have $ab + \delta$, which results in x being computed as $abc + \delta c$. If the value of δ is large enough, i.e., exceeds κ bits, it is possible for x to become longer than r and the value of c be disclosed without protection.

Operations that use the above statistical protection mechanism were implemented as part of SCALE-MAMBA [36] using SPDZ-style computation. Keller et al. [33] report performance of these and other operations in the malicious model with dishonest majority using SPDZ family of protocols, but we were unable to find a separate security analysis of these operations. Note that it is easy to mount additive attacks within SPDZ framework at any point of the computation, i.e., not necessarily during a multiplication, due to the use of additive secret sharing.

As another example, consider generation of a random bit r, using it to protect a private bit b via XOR, and opening $r \oplus b$ during the computation. This protection mechanism achieves perfect secrecy and the disclosure is permitted falling into the first category of functions in Definition 3.1. Using a typical mechanism for random bit generation [8], this computation will result in the following sequence of steps:

(1) $[x] \leftarrow \mathsf{RandFld}()$;
(2) $c \leftarrow \mathsf{Open}([x] \cdot [x])$; (if $c = 0$, restart)
(3) compute e, the smaller square root of c, and its inverse e^{-1};
(4) $[r] \leftarrow ([x]e^{-1} + 1)2^{-1}$;
(5) $y \leftarrow \mathsf{Open}([r] + [b] - 2[r] \cdot [b])$;

The first 4 lines above compute a random bit $[r]$ and the last line computes and opens $r \oplus b$. Similar to the statistical hiding example above, b can be a result of prior computation and be of the form $b_{\text{orig}} + \delta$, where b_{orig} is the true bit value. Then executing line 5 discloses the value of bit b_{orig} to the adversary. In particular, the possibilities for the observed y when $b_{\text{orig}} = 0$ differ from those when $b_{\text{orig}} = 1$ regardless of private bit r when $\delta > 0$. The adversary is also able to mount an additive attack on multiplication on line 2, where the resulting value c can still be a square and thus the computation continues, but the computed r is no longer a bit.

The logic of random bit generation above (i.e., the first 4 lines), modified to work in a ring $\mathbb{Z}_{2^k}$, was shown to be secure in the context of SPD$\mathbb{Z}_{2^k}$ in [14]. Opening of a tampered value on line 2 does not lead to information disclosure because no private inputs are used. However, opening of a private bit protected by a random bit $r \oplus b$ as in the above example is common and thus delaying verification until the end of the evaluation can lead to unintended information disclosure. What is important is that computation over $\mathbb{Z}_{2^k}$ is also vulnerable and attacks are not limited to statistically protected data.

We believe that our analysis has implications on existing implementations and in particular on those that use SPDZ-style delayed verification. Furthermore, this indicates that it may be problematic to separately consider security of building blocks and expect that their composition will remain secure in the presence of delayed verification and calls to Open.

Before we proceed further, we formulate a conservative condition when it is safe to delay verification prior to evaluating an Open gate, which we will use to mark gates which need verification in a preprocessing step. Succinctly, we do not mark gates as needing verification only if they are guaranteed to not reveal any input-dependent information.

Definition 5.1. We refer to any gate, evaluation of which is not secure in the malicious model, as an *attackable gate*. We say that a value is *well-formed* if its computation used no attackable gates.

In the context of this work and based on protocol instantiations in Construction 1, only multiplication and dot product gates are attackable. Based on our analysis, we obtain the following formulation of conditions for delayed verification:

Definition 5.2. An Open gate is guaranteed to be safe without verification of prior computation if (1) the input into the gate is not a function of inputs into the computation f or (2) an input-dependent value is protected via addition with a uniformly random well-formed (per Definition 5.1) field element right before opening.

Clearly, any computation that does not use input-dependent values cannot lead to information leakage (until the computation merges with input-dependent values), which corresponds to case 1. In addition, statistical or perfectly secret protection mechanisms with a one-time pad (i.e. encryption key) drawn from $B \subset \mathbb{F}$ were shown to be vulnerable to leakage attacks as demonstrated above. For that reason, we can only consider protection mechanisms with perfect secrecy and $B = \mathbb{F}$ as not being vulnerable. Furthermore, achieving perfect secrecy with ciphertext space $C = \mathbb{F}$ when the message being protected is potentially tampered and is not guaranteed to be contained in a subspace of $\mathbb{F}$ imposes uniformly distributed keys with key space $\mathcal{K} = \mathbb{F}$ and addition in $\mathbb{F}$ as the encryption operation. Furthermore, we must require that the one-time key is generated according to the specification (i.e., attackable gates were not used) as otherwise it is not guaranteed to be from the desired distribution.

We note that the conditions in Definition 5.2 are sufficient for a safe delay of verification at the time of Open, but they do not necessarily correspond to necessary conditions. In particular, the ability to leak information is influenced by the presence of attackable gates such as multiplications in computing the intermediate result we want to protect as demonstrated in our attacks. This means that attacks are easier for certain types of computation. We leave the question of determining precise bounds as an open problem.

With this in mind, we present Algorithm 1 which marks each open gate with a boolean value V which is 1 if and only if the value to be opened needs verification per Definition 5.2. This algorithm is based on breadth first search and as such has complexity $O(|V|+|E|)$ where the vertex set and edge set correspond to the gates and wires respectively. Given a circuit graph G, we let each gate g store the following attributes, where the first four are based on G and the gate type, while the last two are computed by the algorithm:

(1) Gate in-degree D.
(2) If g is an open gate, output set $B \subseteq \mathbb{F}$ (as per Definition 3.1, and based on function f being computed).
(3) Binary value A, which denotes attackability, as per Definition 5.1.
(4) Binary value R, which is 1 if and only if the gate introduces value into the circuit which is not any parties' protected input (i.e., RandFld, RandInt, or constant gates).
(5) Binary value I, which is 1 if and only if the gate takes input which is dependent on some parties' input.
(6) Binary value V, which denotes whether or not verification needs to take place upon opening.
(7) Each wire w in G will also have a value V, which will match the value $g.V$ associated to the gate for which w is an output wire (this will aid efficiency of Algorithm 1).

Treating the computation circuit G as a directed acyclic graph, this algorithm is comprised of two sequential partial breadth first searches (BFS), which cover G and are mainly disjoint (and hence taken together require only slightly more work than one BFS on G). Thus, the complexity is linear in the number of nodes (gates) and edges. Given that each gate has the fan-out of 1, the complexity is linear in the number of inputs and gates.

The first BFS (lines 2–5) starts at the computation inputs and marks all downstream gates as input-dependent ($I = 1$) and potentially requiring verification ($V = 1$). Then the second BFS (lines 7–20) starts disjoint from the first with gates that contribute input-independent values, i.e., $R = 1$. It runs until any gate marked in the first BFS is encountered, with the goal of identifying input-dependent open gates which use well-formed randomness for protection and re-setting $V = 0$ in such cases.

Algorithm 1 Open Gate Categorization Algorithm

Input: Graph representation of circuit G with gates as vertices and wires as edges, with associated values D, A, B, and R.

Output: Binary value V for each open gate g that denotes whether verification needs to take place prior to opening.

Protocol steps:

1: For each gate g and wire w, set $g.V = 0$ and $w.V = 0$
2: Create dummy gate g_0 that outputs to each input wire in G
3: Run a BFS starting at g_0
4: **where**
5: When visiting gate g via wire w, set $g.I = g.V = w.V := 1$
6: Set $O_i = \{\}$ and $O_r = \{\}$
7: Create dummy gate g_r that outputs to each gate g with $g.R = 1$
8: Run a BFS starting at g_r
9: **where** for each gate g, discovered via wire w_{in},
10: **if** g is an addition gate, not marked as finished, with input wires w_a and w_b and $g.I = 1$ **then**
11: **if** g outputs via wire w_c into open gate h **then**
12: Add (w_a, w_b, g, w_c, h) to O_i
13: **else**
14: **if** g is an open gate with $g.I = 0$ **then**
15: Add g to O_r
16: Set $g.V = (g.V$ **or** $g.A$ **or** $w_{\text{in}}.V)$
17: Decrement $g.D$
18: **if** $(g.D = 0)$ **or** $(g.V = 1)$ **then**
19: Set $w_{\text{out}}.V = g.V$ for all w_{out} which are (directly connected to) output of g
20: Mark g as finished, set $g.D = 0$, and if $g.I = 0$, then process its downstream neighbors
21: **for** each $(w_a, w_b, g, w_c, h) \in O_i$ **do**
22: $h.V := (w_a.V$ **and** $w_b.V)$ **or** $(h.B \overset{?}{\neq} \mathbb{F})$
23: **for** each $g \in O_r$ **do**
24: $g.V := 0$

During the second BFS, whenever an input-dependent addition gate is found to output into an open gate, the pair of gates is added along with wires as an atomic unit to a list O_i (lines 10–12). This is a list of potential input-dependent open gates which are safe to open, per condition (2) of Definition 5.2. We create the list O_i in order to defer evaluation until all upstream gates have been considered.

Any open gate *not* downstream of input is added to another list O_r (lines 14–15), which maintains the list of all input-independent open gates. The gates in this category do not require verification (per condition (1) of Definition 5.2), but may be temporarily marked by the algorithm as needing verification in order to correctly process condition 2, i.e., determine if a random pad using such value is well-formed. The need-to-verify property of each gate is assigned on line 16, inclusive of g's attackability, as well as all upstream and local need-to-verify properties. Consequently, the assignment on line 19 propagates this flag downstream.

In the final step, this flag is cleared (i.e., set $V = 0$) for all input-independent open gates in O_r (lines 23–24), as well as any-input dependent open gate in O_i which has been padded by a well-formed uniform random element of $\mathbb{F}$ immediately prior to opening (lines 21–22). The former corresponds to condition (1) of Definition 5.2 and the latter corresponds to condition (2).

Lemma 5.3 states correctness, with the proof in Appendix B.

Lemma 5.3. *For a given circuit G and open gate h, Algorithm 1 labels h as safe to open without verification if and only if it conforms to Definition 5.2.*

5.2 From Security up to Additive Attacks to Security with Abort

Following [9], our transformation of building blocks secure up to additive attacks into an actively secure construction is given in the hybrid model. For that reason we need to define a number of ideal functionalities (and have protocols that realize the ideal functionalities). A number of them, $\mathcal{F}_{\text{mult}}$, $\mathcal{F}_{\text{dotprod}}$, $\mathcal{F}_{\text{randfld}}$, $\mathcal{F}_{\text{randint}}(k)$, and $\mathcal{F}_{\text{open}}$, correspond to sub-protocols for different types of gates, with the adversary being able to introduce an additive error in $\mathcal{F}_{\text{mult}}$ and $\mathcal{F}_{\text{dotprod}}$ and others being resilient to malicious behavior. There are also functionalities for entering input into the computation, $\mathcal{F}_{\text{input}}$, and delivering output to a party, $\mathcal{F}_{\text{output}}$, together with two supplemental functionalities $\mathcal{F}_{\text{randfldpub}}$ and $\mathcal{F}_{\text{zerocheck}}$ modeled similar to [9]. $\mathcal{F}_{\text{randfldpub}}$ generates a publicly known random field element and is instantiated by calling $\mathcal{F}_{\text{randfld}}$ and $\mathcal{F}_{\text{open}}$; and $\mathcal{F}_{\text{zerocheck}}$ checks whether a private value equals to 0 and is instantiated by randomizing the argument (calling $\mathcal{F}_{\text{randfld}}$ and $\mathcal{F}_{\text{mult}}$), opening it (using $\mathcal{F}_{\text{open}}$) and comparing the opened value to 0. The details of the ideal functionalities are given in Appendix C.

Our main compiler that converts components secure up to additive attacks into a construction secure in the presence of active adversaries is Construction 2. The computation is carried out by n participants, and as before there are m inputs for any m. It assumes a large field, i.e., $|\mathbb{F}|$ is on the order of 2^κ for a security parameter κ. (The solution can be generalized to work with smaller fields as shown later.) With sufficiently large fields, there are two parallel branches of computation: (1) the regular computation and (2) a randomized computation, where all values are multiplied by a random field element r not known to the parties during the computation. When the current gate prescribes opening, we open the value only in the regular (non-randomized) branch. When a gate generates random value $[z]$, we create the corresponding value $[r \cdot z]$ for the second execution. Every time verification is triggered (prior to an opening marked as requiring verification or prior to reconstructing the output), we verify consistent execution of all attackable gates (multiplication and dot product) since the last verification operation. If verification succeeds, the parties continue.

Additionally, when opening values which are not a function of any parties' input on some proper subset $B \subset \mathbb{F}$, we allow for parties to perform checking that the opened value is in B and abort if necessary, provided an efficient checking algorithm is known to the parties. This is not necessary to uphold security since no sensitive data was leaked and the adversary has not obtained any more information relative to the honest parties' view than they would when tampering an unopened shared value. Moreover it does not guarantee that there is no tampering since there is always a chance an additive attack maps to another value in B. Nevertheless, this can with some likelihood provide a relatively efficient way to detect misbehavior prior to scheduled verification.

Construction 2. *Secure Function Evaluation Over Large Fields. If any participant receives $\perp$ from any sub-functionality throughout the protocol execution (e.g., $\mathcal{F}_{\text{input}}$, $\mathcal{F}_{\text{mult}}$, etc.), it sends $\perp$ to all other parties, outputs $\perp$, and terminates.*

(1) Input sharing: Each input owner sends its input v to $\mathcal{F}_{\text{input}}$, which results in party P_i learning its share $[v]_i$.
(2) Generation of circuit randomization: The parties call $\mathcal{F}_{\text{randfld}}$ and receive $[r]$. Each party initializes a verification buffer to hold values for Step 4h.
(3) Randomization of inputs: For each input $[v]$, the parties call $\mathcal{F}_{\text{mult}}$ on inputs $[r]$ and $[v]$ and obtain $[rv]$. The parties store $([v], [rv])$ and add the tuple to the verification buffer.
(4) Function evaluation: The parties are given the function as a sequence of gates in a predetermined topological order and evaluate it as follows based on the type of the gate:
  (a) Addition: on input $([x], [rx])$ and $([y], [ry])$, each participant locally computes $[z] = [x + y]$, $[rz] = [rx] + [ry]$ using its shares and stores $([z], [rz])$.
  (b) Multiplication by a known field element: on input $([x], [rx])$ and $y \in \mathbb{F}$, each party locally computes $[z] = y \cdot [x] = [xy]$, $[rz] = y \cdot [rx] = [rxy]$ using its shares and stores $([z], [rz])$.
  (c) Multiplication: on input $([x], [rx])$ and $([y], [ry])$, the parties call $\mathcal{F}_{\text{mult}}$ on $[x]$ and $[y]$ to obtain $[z] = [xy]$ and on $[rx]$ and $[y]$ to obtain $[rz] = [rxy]$. They store $([z], [rz])$ and add the tuple to the verification buffer.
  (d) Dot product: on input $\{([x_i], [rx_i])\}_{i=1}^{\ell}$ and $\{([y_i], [ry_i])\}_{i=1}^{\ell}$, the parties call $\mathcal{F}_{\text{dotprod}}$ on $\{[x_i]\}$ and $\{[y_i]\}$ to obtain $[z] = \sum_i [x_i y_i]$ and also on $\{[rx_i]\}$ and $\{[y_i]\}$ to obtain $[rz] = \sum_i [rx_i y_i]$. The parties store $([z], [rz])$ and add the tuple to the verification buffer.
  (e) RandFld gate: the parties call $\mathcal{F}_{\text{randfld}}$ to obtain $[w]$, then call $\mathcal{F}_{\text{mult}}$ on $[r]$ and $[w]$ to obtain $[rw]$, store $([w], [rw])$, and add the tuple to the verification buffer.
  (f) RandInt gate: the parties call $\mathcal{F}_{\text{randint}}$ to obtain $[w]$, then call $\mathcal{F}_{\text{mult}}$ on $[r]$ and $[w]$ to obtain $[rw]$, store $([w], [rw])$, and add the tuple to the verification buffer.
  (g) Opening: if the gate is marked by Algorithm 1 as needing verification, the parties proceed to verification in Step 4h. If the verification did not abort or if the gate is not marked as needing verification, the parties use input $([x], [rx])$ and call $\mathcal{F}_{\text{open}}$ on $[x]$ to receive x. If efficient testing of membership in B exists and the parties find that $x \notin B$, the parties abort. If no abort has occurred, the parties store x.
  (h) Verification of evaluation: Let pairs $([x_i], [rx_i])_{i=1}^{M}$ denote values currently stored in the verification buffer. Verification proceeds as follows:
    (i) If $M > 1$, the participants call $\mathcal{F}_{\text{randfldpub}}$ to generate a random seed s and make M calls to $\mathbb{F}\text{PRG}(s)$ to determine pseudo-random $\alpha_1, \ldots, \alpha_M$. Otherwise, set $\alpha_1 = 1$.
    (ii) Each party locally computes on its shares $[u_1] = \sum_{i=1}^{M} \alpha_i \cdot [rx_i]$ and $[u_2] = \sum_{i=1}^{M} \alpha_i \cdot [x_i]$
    (iii) The parties call $\mathcal{F}_{\text{mult}}$ to obtain $[r] \cdot [u_2]$ and compute $[T] = [u_1] - [r] \cdot [u_2]$.
    (iv) The participants call $\mathcal{F}_{\text{zerocheck}}$ on $[T]$; if $\mathcal{F}_{\text{zerocheck}}$ outputs reject, the parties output $\perp$ and terminate (and otherwise $\mathcal{F}_{\text{zerocheck}}$ outputs accept and the parties continue).
    (v) If no abort occurred, the parties flush the verification buffer.
(5) Output reconstruction: The parties run one final round of verification as per Step 4h. If no abort occurred, the parties call $\mathcal{F}_{\text{output}}$ to reconstruct each outputs x to the party receiving it.

To demonstrate security, as the first step we prove that if an adversary misbehaves during the computation of any multiplication or dot product, then the probability that the verification step does not detect this is negligible in the security parameter.

Theorem 5.4. *If during a call to any $\mathcal{F}_{\text{mult}}$ or $\mathcal{F}_{\text{dotprod}}$ in Construction 2 the adversary introduces additive error $d \neq 0$, then T equals 0 with probability less than $2/|\mathbb{F}|$.*

The proof is deferred to Appendix C. Now, we state overall security of Construction 2 as follows:

Theorem 5.5. *Let $\mathbb{F}$ be a large finite field such that $1/|\mathbb{F}|$ is negligible in statistical security parameter κ. Then for any function f over $\mathbb{F}$ complying with Definition 3.1, Construction 2 securely evaluates f with abort in the presence of malicious adversaries controlling at most $t < n/2$ participants in the $(\mathcal{F}_{\text{input}}, \mathcal{F}_{\text{output}}, \mathcal{F}_{\text{open}}, \mathcal{F}_{\text{randfld}}, \mathcal{F}_{\text{randint}}, \mathcal{F}_{\text{randfldpub}}, \mathcal{F}_{\text{mult}}, \mathcal{F}_{\text{dotprod}}, \mathcal{F}_{\text{zerocheck}})$-hybrid model. In the absence of malicious behavior the computation always produces the correct result, while in the case of tampering with the computation, if the computation successfully terminates, the output is incorrect with at most negligible probability in κ.*

The proof is provided in Appendix C. We also discuss modifying the solution to work with smaller fields (by performing a number of randomized executions) in Appendix D.

6 PERFORMANCE

We implemented our protocols[1] and evaluate their performance in a three-party setting. All experiments use identical 2.4GHz virtual machines with 26GB of RAM. They were connected via 10Gbps Ethernet links, which we throttled to 1Gbps using the tc command. Two-way latency was measured to be 0.106 ms. All experiments are single threaded.

We evaluate performance of different functionalities in both semi-honest and malicious models. We are not aware of prior work along this line of research drawing an empirical comparison between semi-honest and transformed malicious variants. Inevitably, the added cost of maliciously secure protocols would differ depending on the function being evaluated because of additional work associated with each input and the gates of the computation themselves.

We run both micro-benchmarks with individual operations as well as offer evaluation of a more complex function, namely over-the-threshold Euclidean distance, as listed in Table 1.

Micro-benchmarks include addition Add (which is a special case and the reason for its inclusion is discussed later), multiplication Mult, equality EQ as specified in [8], and less than LT, also from [8]. As we strive to measure the difference in performance when we enhance security from the semi-honest to the malicious model, we

[1]The implementation is available at https://github.com/applied-crypto-lab/mal-ext-circuits.

Protocol	Setup	Field size	Runtime of different batch sizes (in ms)						Communication per party (in elem)
			1	10	10^2	10^3	10^4	10^5	
Add	SH	32	0.001	0.001	0.004	0.041	0.371	3.35	0
	M	48	0.000	0.01	0.008	0.088	0.859	8.14	0
	VC		0%	0%	0%	0%	0%	0%	
Mult	SH	32	0.069	0.074	0.186	1.32	10.8	104	1 per op.
	M	48	0.264	0.943	1.22	4.28	32.98	335	2 per op. + 6
	VC		75.9%	89.8%	73.6%	37.2%	19.6%	17.7%	
LT	SH	80	0.515	2.78	21.8	226	2,292	23,014	220 per op.
	M	80	3.95	17.7	143	1,349	13,418	N/A	536 per op. + 18
	VC		53.3%	27.0%	14.7%	12.6%	12.2%	N/A	
EQ	SH	80	0.444	1.80	13.6	125	1,283	12,867	100 per op.
	M	80	3.11	11.2	69.7	694	6,576	65,165	249 per op. + 12
	VC		59.3%	28.8%	14.8%	12.3%	11.4%	11.3%	
Over-the-threshold Euclidean distance	SH	80	1.03	1.08	1.10	1.22	1.73	10.8	221
	M	80	3.98	4.00	4.01	4.57	6.16	23.4	554
	VC		52.1%	52.2%	52.1%	50.2%	36.6%	10.0%	

Table 1: Performance of different functionalities implemented as extended circuits.

implemented the computation in both models using PICCO [40] that implements multiplication [5] that Construction 1 uses.

In Table 1, notation SH stands for "semi-honest," M for "malicious," and VC for "verification cost" as a percentage of execution time in the malicious model. All functionalities use 32-bit integer inputs. Field size denotes the bitlength of each field element throughout the computation. Certain operations such as comparisons involve statistical hiding and increase the bitlength of field elements by a statistical parameter. We set all statistical security parameters to 48. Computation size in individual operations denotes how many operations were executed at the same time in a single batch, while it means the size of the input for more involved functions. In addition, we report the fraction of the time spent on verification in maliciously secure protocols. Input randomization in the malicious model is excluded as it is a one-time cost and has small impact on individual operations. Communication is measured as the number of field elements sent by each party. For example, the cost of performing 10^3 multiplications in Table 1 is 1,000 elements or 4,000B per party in the semi-honest setting and 2,006 elements or 12,036B per party in the malicious model.

Overall, the runtime of malicious experiments for our computation is often 3–4 times slower compared to the corresponding experiment in the semi-honest model. The difference is higher for a single operation, where the added round complexity of verification dominates the cost and decreases as the computation size increases. The main factor that introduces the extra cost in the malicious experiments for non-tiny computations is the randomized branch, which doubles communication and computation. Communication can increase beyond the factor of two because some gates (e.g., RandFld) are local prior to transformation, but their output needs to be randomized, which uses communication. The time can also increase for functionalities that move to a larger field size after the transformation.

An invocation of the verification procedure in the malicious experiments normally involves three rounds of communication. It is invoked (according to Algorithm 1) once in Mult experiments, twice in EQ experiments, and three times in LT experiments. ("VC" corresponds to the aggregate time of all calls.)

It is also informative to compare the cost of semi-honest to malicious security transformation to that in other publications. While the differences in the hardware and implementation choices prevent a direct comparison of previously published runtimes, we examine communication cost. In [19], Escudero et al. implement an edaBit-based LT solution, which in a comparable setting uses 1.4Kb per operation in the semi-honest model, which rises to 19.9Kb in the malicious setting. Our implementation starts with 17.8Kb in the semi-honest model and rises to 45.4Kb in the malicious model. This points to the opportunity of the two works to complement each other: our work can benefit from integration of more efficient random bit generation, while [19] and other publications that rely on delayed verification to achieve malicious security can use our work to determine when it is safe to open intermediate results in a protocol without triggering verification.

To develop a better understanding of the benefits of using extended circuits compared to pure arithmetic circuits composed of addition and multiplication gates, we also implement a semi-honest and its transformed malicious protocol composed of strictly arithmetic gates on the example of the over-the-threshold Euclidean distance (as described in the beginning of this work). This computation uses additions and multiplications followed by a comparison operation. As discussed in the beginning of the paper, the presence of non-arithmetic operations such as equality and greater-than comparisons requires the entire computation to be carried out in the bit-decomposed form on Boolean values when the circuit is composed of two types of arithmetic gates (and significantly increases the cost of integer additions and multiplications). Thus, we implement Add, Mult, EQ, LT, and the entire over-the-threshold Euclidean distance on bit-decomposed (binary) values using Boolean gates and report performance on 32-bit inputs in Table 2.

Protocol	Mult. gates per op.	Setup	Field size	Runtime of dif. batch sizes (in ms)				Communication per party (in elem)
				1	10	10^2	10^3	
Add	192	SH	3	0.556	2.90	23.4	267	192 per op.
		M	48	1.78	8.17	58.1	661	384 per op. + 6
		VC		50.9%	26.6%	18.4%	15.5%	
Mult	3695	SH	3	7.04	50.3	555	5,429	3695 per op.
		M	48	16.4	123	1,321	13,245	7390 per op. + 6
		VC		19.8%	16.0%	15.0%	13.7%	
LT	94	SH	3	0.471	1.64	12.2	128	94 per op.
		M	48	1.47	4.74	32.7	324	188 per op. + 6
		VC		57.9%	31.9%	18.3%	15.2%	
EQ	63	SH	3	0.435	1.29	8.77	84.8	63 per op.
		M	48	1.40	3.58	23.1	228	126 per op. + 6
		VC		61.3%	33.9%	19.7%	14.6%	
Over-the-threshold Euclidean distance	$4270\ell - 98$	SH	3	8.52	59.9	623	6,315	$4270\ell - 98$
		M	48	19.5	153	1,524	15,299	$8540\ell - 92$
		VC		17.0%	15.4%	15.0%	13.5%	

Table 2: Performance of different functionalities implemented using strict arithmetic circuits (bitwise computation). ℓ denotes the size of the input sequences in the case of over-the-threshold Euclidean distance.

We used optimized implementations of integer operations over field $\mathbb{F} = \mathbb{Z}_5$ in the semi-honest model.[2] The bitwise (binary) circuits for addition, EQ, and LT have logarithmic in the length of the arguments number of communication rounds, comparisons (LT and EQ) have linear complexity, while addition has complexity $O(k \log k)$. The bitwise addition and LT protocols can be found in [38]. For 32-bit operands, a logarithmic number of rounds is comparable to the number of rounds in constant-round comparison protocols.

We also implemented efficient bitwise multiplication and summation which each call this optimized bitwise addition in a number of rounds logarithmic in the argument bit length. Both algorithms fan in as many pairs as possible each round while maximizing the number of values which can be passed though on each round (i.e., minimizing the operand length). We note that overall these algorithms have log squared round complexity in the length of the arguments. Bitwise multiplication is used for the Mult experiments as well as the batch multiplication as part of the over-the-threshold Euclidean distance in Table 2, while the bitwise summation is used to sum over the resulting products in the latter computation.

If we look at individual operations, bitwise comparisons in Table 2 are faster than comparisons in Table 1, while bitwise addition and multiplication in Table 2 are, as expected, substantially slower than the corresponding operations in Table 1 that do not require computation in a bit-decomposed form. (Local) addition Add was added to Table 1 to show the difference in performance between the two variants (note that because Add experiments in Table 1 do not include any attackable gates, verification is not used).

When we combine the building blocks to implement the over-the-threshold Euclidean distance, we observe that bitwise variants (strict arithmetic circuit) are one or more orders of magnitude slower than the variants that use the extended set of gates. A similar performance difference is expected for certain types of popular computations such as privacy-preserving neural network inference that makes a heavy use of multiplications and comparisons.

[2]The secret sharing scheme requires that the field contains at least $n + 1$ elements, which prevents the use of $\mathbb{Z}_2$.

7 CONCLUSIONS

In this work we study extending the notion of semi-honest protocols secure up to additive attacks to cover an extended set of elementary gates beyond conventional arithmetic circuits composed of additions and multiplications. The extended set includes randomization and reconstruction gates which create challenges when working on this problem. We first define constraints on functions, evaluation of which can be simulated in the presence of open gates. We then proceed with extending the notion of security up to additive attacks to cover the additional gates which, according to the requirements of the framework, cannot contain open gates. Our consequent analysis shows that delayed verification is not always safe in the presence of open gates and devise an algorithm for triggering verification early when opening a potentially tampered value is not considered safe. Our final result is transformation to achieve security in the malicious model with abort and empirically demonstrate that the difference in performance of semi-honest and malicious variants is low. Furthermore, for certain functions the computation is significantly more efficient than using conventional arithmetic circuits.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their feedback. This work was supported in part by a Google Faculty Research Award, Buffalo Blue Sky Initiative, and US National Science Foundation grant 2213057. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding sources.

REFERENCES

[1] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. 2021. An efficient passive-to-active compiler for honest-majority MPC over rings. , 122–152 pages.

[2] Judit Bar-Ilan and Donald Beaver. 1989. Non-Cryptographic Fault-Tolerant Computing in Constant Number of Rounds of Interaction. In *ACM Symposium on Principles of Distributed Computing*. 201–209.

[3] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology – CRYPTO*. 420–432.

[4] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The Round Complexity of Secure Protocols. In *ACM Symposium on Theory of Computing (STOC)*. 503–513.

[5] Marina Blanton, Ahreum Kang, and Chen Yuan. 2020. Improved Building Blocks for Secure Multi-Party Computation based on Secret Sharing with Honest Majority. In *International Conference on Applied Cryptography and Network Security (ACNS)*. 377–397.

[6] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *European Symposium on Research in Computer Security (ESORICS)*. 192–206.

[7] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In *Advances in Cryptology – CRYPTO*. 67–97.

[8] Octavian Catrina and Sebastiaan de Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *International Conference on Security and Cryptography for Networks (SCN)*. 182–199.

[9] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast large-scale honest-majority MPC for malicious adversaries. In *Advances in Cryptology – CRYPTO*. 34–64.

[10] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *Advances in Cryptology – CRYPTO*. 769–798.

[11] Ronald Cramer, Ivan Damgård, and Yuval Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *Theory of Cryptography Conference (TCC)*. 342–362.

[12] Ronald Cramer, Yevgeniy Dodis, Serge Fehr, Carles Padró, and Daniel Wichs. 2008. Detection of Algebraic Manipulation with Applications to Robust Secret Sharing and Fuzzy Extractors. In *Advances in Cryptology – EUROCRYPT*. 471–488.

[13] Anders P.K. Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security.. In *USENIX Security Symposium*. 2183–2200.

[14] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *IEEE Symposium on Security and Privacy*. 1102–1120.

[15] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. 2006. Unconditioannly Secure Constant-Rounds Multi-Party Computation for Equality, Comparison, Bits and Exponentiation. In *Theory of Cryptography Conference (TCC)*. 285–304.

[16] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and unconditionally secure multiparty computation. In *Advances in Cryptology – CRYPTO*. 572–590.

[17] Ivan Damgård and Rune Thorbek. 2007. Non-interactive proofs for integer multiplication. In *Advances in Cryptology – EUROCRYPT*. 412–429.

[18] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology – CRYPTO*. 643–662.

[19] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *Advances in Cryptology – CRYPTO*. 823–852.

[20] Jun Furukawa and Yehuda Lindell. 2019. Two-Thirds Honest-Majority MPC for Malicious Adversaries at Almost the Cost of Semi-Honest. In *ACM Conference on Computer and Communications Security (CCS)*. 1557–1571.

[21] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. 2014. Circuits resilient to additive attacks with applications to secure computation. In *ACM Symposium on Theory of Computing (STOC)*. 495–504.

[22] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. 2015. Circuits resilient to additive attacks with applications to secure computation. IACR Cryptology ePrint Archive Report 2015/154.

[23] Daniel Genkin, Yuval Ishai, and Mor Weiss. 2016. Binary AMD Circuits from Secure Multiparty Computation. In *International Conference on Theory of Cryptography (TCC)*. 336–366.

[24] Rosario Gennaro, Michael Rabin, and Tal Rabin. 1998. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 101–111.

[25] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. 2021. ATLAS: Efficient and scalable MPC in the honest majority setting. In *Advances in Cryptology – CRYPTO*. 244–274.

[26] Vipul Goyal and Yifan Song. 2020. Malicious Security Comes Free in Honest-Majority MPC. IACR Cryptology ePrint Archive Report 2020/134.

[27] Vipul Goyal, Yifan Song, and Chenzhi Zhu. 2020. Guaranteed output delivery comes free in honest majority MPC. In *Advances in Cryptology – CRYPTO*. 618–646.

[28] Philip Hall. 1927. The Distribution of Means for Samples of Size N Drawn from a Population in which the Variate Takes Values Between 0 and 1, All Such Values Being Equally Probable. *Biometrika* 19, 3–4 (1927), 240–245.

[29] Carmit Hazay, Abhi Shelat, and Muthuramakrishnan Venkitasubramaniam. 2020. Going Beyond Dual Execution: MPC for Functions with Efficient Verification. In *Public-Key Cryptography (PKC)*. 328–356.

[30] Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, and Mor Weiss. 2020. The Price of Active Security in Cryptographic Protocols. In *Advances in Cryptology – EUROCRYPT 2020*. 184–215.

[31] J.O. Irwin. 1927. On the Frequency Distribution of the Means of Samples from a Population Having any Law of Frequency with Finite Moments, with Special Reference to Pearson's Type II. *Biometrika* 19, 3–4 (1927), 225–239.

[32] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* (1989), 56–64.

[33] Marcel Keller, Peter Scholl, and Nigel P. Smart. 2013. An architecture for practical actively secure MPC with dishonest majority. In *ACM Conference on Computer and Communications Security (CCS)*. 549–560.

[34] Yehuda Lindell and Ariel Nof. 2017. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest majority. In *ACM Conference on Computer and Communications Security (CCS)*. 259–276.

[35] Peter Sebastian Nordholt and Meilof Veeningen. 2018. Minimising communication in honest-majority MPC by batchwise multiplication verification. In *Applied Cryptography and Network Security (ACNS)*. 321–339.

[36] scale-mamba-doc 2021. SCALE-MAMBA Documentation. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation-SCALE.pdf.

[37] scale-mamba-imp 2021. SCALE-MAMBA Implementation. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[38] SecureSCM. 2009. Deliverable D9.2, EU FP7 Project Secure Supply Chain Management (SecureSCM). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.221.393&rep=rep1&type=pdf.

[39] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* (1979), 612–613.

[40] Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: A General-Purpose Compiler for Private Distributed Computation. In *ACM Conference on Computer and Communications Security (CCS)*. 813–826.

A WEAKLY PRIVATE BUILDING BLOCKS

Lemma A.1. *The protocol instantiations for each gate in Construction 1 correctly realize the corresponding functionalities.*

Proof. We examine each gate in turn:

- RandFld gate: During evaluation of this gate, each party P_i, $i \in [1, n]$, locally computes:

$$[x]_i = \sum_{Q \in \mathcal{Q}, i \in Q} \mathbb{F}\mathsf{PRG}(\mathsf{key}_Q) \cdot \lambda_Q(i) \tag{1}$$

  Recall that for each maximal unqualified set T of the threshold access structure ($|T| = t$), $Q = [1, n] \setminus T$ and the corresponding share of x, x_Q, is given to each party in Q. Also recall that λ_Q was defined as a unique polynomial of degree t such that $\lambda_Q(0) = 1$ and $\lambda_Q(i) = 0$ for each $P_i \notin Q$. This means that shares of all parties not in Q in equation 1 will be equal to 0 and cannot contribute. As explained in [11], we can express the result of this operation as

$$x = \sum_{Q \subset [1,n], |Q| = n-t} \mathbb{F}\mathsf{PRG}(\mathsf{key}_Q),$$

  which is a pseudo-random element of the field. Then the polynomial

$$f = \sum_{Q \subset [1,n], |Q| = n-t} \mathbb{F}\mathsf{PRF}(\mathsf{key}_Q) \cdot \lambda_Q(i)$$

  has degree t, $f(0) = x$, and $f(i) = [x]_i$ as desired.

- RandInt gate: During evaluation of this gate on parameter k, the computation performed by party P_i, $i \in [1, n]$, is defined as:
$$[x]_i = \sum_{Q \in \mathcal{Q}, i \in Q} \mathbb{F}\mathsf{PRG}(\mathsf{key}_Q) \cdot \lambda_Q(i)$$
Similar to the case of RandFld gate, secret shared x now corresponds to the sum of pseudo-random integers
$$x = \sum_{Q \subset [1,n], |Q| = n-t} \mathbb{Z}\mathsf{PRG}_k(\mathsf{key}_Q),$$
but unlike RandFld, the sum combines k-bit integers and the result is $k + \log(\nu)$ bits long, where $\nu = \binom{n}{t}$ is the number of shares.
- Addition gate: Correctness of local addition of shares follows from the linear property of the secret sharing scheme.
- Multiplication by a known field element gate: Correctness of a local multiplication of shares by a known field element follows from the linear property of the secret sharing scheme.
- Multiplication gate (linear communication): During the protocol, the parties first compute $\langle v \rangle$ and open the value. The result is consequently set to $[z] = v - [w]$ and we want to show that $z = a \cdot b$.
The rationale behind the protocol is that computing the product locally using shares raises the polynomial degree and degree reduction is performed via generating a pair $[w], \langle u \rangle$ that correspond to the same value ($w = u$), but are encoded as degree-t and degree-$2t$ polynomials, respectively. Thus, the parties compute $\langle v \rangle$ as a sum of a pseudo-random w and the product $a \cdot b$, represented as a polynomial of degree $2t$. Thus, after reconstructing v, we obtain $z = v - w = w + a \cdot b - w = a \cdot b$ as desired.
- Multiplication gate (single round): As in the case of the previous multiplication protocol, each party locally computes $\langle z \rangle_i = [a]_i [b]_i$, but the degree reduction, or re-sharing, mechanism to obtain $[z]$ is different. Each P_i re-shares its $\langle z \rangle_i$ as a degree t and polynomial interpolation constants α_js are used to combine the shares and obtain degree-t shares of z. The constants α_js are designed in [24] to reconstruct the shares of the product $z = a \cdot b$ as $[z]_i = \sum_{j=1}^{n} \alpha_j [d_j]_i$, where $[d_j]_i$ denotes party P_i's share of $\langle z \rangle_j$. This is the same reconstruction as what our protocol uses.
However, instead of setting the coefficients of the polynomial f that represents secret sharing of $\langle z \rangle_i$, denoted as $f_{\langle z \rangle_i}$, at random (with the requirement that $f_{\langle z \rangle_i}(0) = \langle z \rangle_i$), P_i uses t pseudo-random values from which the polynomial is reconstructed (together with the value $f_{\langle z \rangle_i}(0) = \langle z \rangle_i$) as originally described in [5]. This does not change the fact that $f_{\langle z \rangle_i}$ corresponds to a valid representation of sharing of $\langle z \rangle_i$, but only lowers communication overhead that reduces the number of shares that need to be communicated to other parties from $n - 1$ to $n - t - 1$. Thus, this does not impact correctness.
- Dot product gate: The reasoning behind the dot product protocols is similar to that of multiplication gates above. The only difference is that instead of computing (degree-$2t$) shares of the product $\langle z \rangle_i = [x]_i [y]_i$, each party P_i computes share $\langle z \rangle_i = \sum_{j=1}^{\ell} [x_j]_i \cdot [y_j]_i$, which also corresponds to a polynomial of degree $2t$. The rest of the computation is unchanged and correctness follows.

□

Proof of Theorem 4.4. Given simulator $\mathcal{S}$ and adversary $\mathcal{A}$ which controls the set of corrupt parties C, the construction is simulated as described next. Since there is no cause for honest parties to abort during the course of Construction 1, we do not need $\mathcal{S}$ to maintain a complete view of computation, and instead will focus only on protocols where communication would occur.

(1) *Input secret sharing:* $\mathcal{S}$ calls share(0) for each element of P_i's input x_i, where $P_i \in H$, and distributes them according to the construction. It also receives shares received from the malicious parties.
(2) *Function evaluation:* For each gate g of the following type, in the given topological order,
- *Multiplication gate (linear communication)*:
  (a) For each honest party $P_i \in H$, $\mathcal{S}$ defines a sharing $[r]$ by generating random shares using RSS, and similarly generates a fresh $\langle 0 \rangle$.
  (b) For each honest party $P_i \in H$, $\mathcal{S}$ computes $\langle u \rangle_i = [r]_i \cdot [1]_i + \langle 0 \rangle_i$.
  (c) For each honest party $P_i \in H$, $\mathcal{S}$ computes $\langle D \rangle_i = [x]_i \cdot [y]_i + \langle R \rangle_i$;
  (d) If the king is honest:
    (i) Acting as the king, $\mathcal{S}$ receives $\langle D \rangle_j$ from each corrupt party P_j.
    (ii) Acting as the king, $\mathcal{S}$ reconstructs D and sends D to the corrupted parties.
  Otherwise, if the king is corrupt:
    (i) For each honest party $P_i \in H$, $\mathcal{S}$ sends $\langle D \rangle_i$ to the corrupt king;
    (ii) For each honest party $P_i \in H$, $\mathcal{S}$ receives D from the corrupt king;
- *Multiplication gate (single round)*:
  (a) For each honest party $P_i \in H$ and for each $j = 1 + (i \bmod n), \ldots, 1 + ((i + t) \bmod n)$, $\mathcal{S}$ computes $[d_i]_j = \mathbb{F}\mathsf{PRG}(k_{j,i})$.
  (b) For each honest party $P_i \in H$, $\mathcal{S}$ computes $\langle z \rangle_i = [x]_i \cdot [y]_i$ and sets $[d_i]_0 = \langle z \rangle_i$.
  (c) For each honest party $P_i \in H$, $\mathcal{S}$ reconstructs the polynomial $f_{\langle z \rangle_i}$ from the $t + 1$ shares.
  (d) For each honest party $P_i \in H$, $\mathcal{S}$ evaluates and sends $[d_i]_j = f_{\langle z \rangle_i}(j)$ to P_j for $P_j \in C$ such that $j = 1 + ((i + t + 1) \bmod n), \ldots, 1 + ((i + n - 2) \bmod n)$
- *Dot product gate*: For each honest party $P_i \in H$, $\mathcal{S}$ computes $\langle c \rangle_i = \sum_{j=1}^{\ell} [x_j] \cdot [y_j]$ and either
  – Stores $\langle z \rangle_i = \langle c \rangle_i$ and proceeds with single round multiplication in Step 2c using this value.
  – Stores $[x]_i \cdot [y]_i = \langle c \rangle_i$ and depending on the honesty of the king, proceeds with linear communication multiplication in Step 2(d)i of the relevant branch using this value.

Next, we prove the indistinguishability of the real and simulated views. Note that the only difference between a real-world execution and a simulation is that we let $\mathcal{S}$ take zeros as honest parties' input.

But due to the properties of the underlying secret sharing scheme, a dishonest minority cannot learn any information about input values shared by any party or intermediate shares that depend on those values. This is independent of the party performing the sharing, including any simulator on behalf of honest parties.

Now we consider building blocks that involve interactive operations. For these, $\mathcal{A}$ can tamper with operations by engaging in different types of misbehavior. These include

- Modifying intermediate results (which can influence the values sent to honest parties).
- Create shared values using an incorrect threshold (i.e., create polynomials of a wrong degree).
- Sending different values to different honest parties when the protocol calls for broadcasting the same value of a number of participants.

The first place that $\mathcal{A}$ can tamper with is input sharing phase, where the corrupt parties may send shares with wrong degree to $\mathcal{S}$. We argue that incorrect degree sharing can only affect the correctness, and not privacy. To that end, note that because inputs have to go though some arithmetic gates before being opened, if inputs are shared by some corrupt parties with degree lower than t, then the degree will become t in the next arithmetic gate that the shares combine with any other inputs or randomness. If instead some corrupt parties share input with degree greater than t, then the degree will become t in the next multiplication gate. Hence, there will be no distinguishable difference between the real and simulated views if $\mathcal{A}$ improperly shares values.

In the simulation of the single round multiplication gate, only simulation step 2d involves interactive operations. $\mathcal{A}$ may tamper with this protocol by sending honest parties incorrect shares. Note that the above discussion on incorrect degree applies to this situation. Otherwise, sending incorrect shares of correct degree can only affect output correctness, and not privacy. Hence, the real and simulated views remain indistinguishable.

In the simulation of the linear communication multiplication gate, we consider two cases. If the king is honest, then $\mathcal{A}$ may send wrong shares to $\mathcal{S}$ in step 2(d)i. As previously noted, this manner of misbehavior will not cause a distinguishable difference between the real and simulated views. If instead the king is corrupt, $\mathcal{A}$ may broadcast an incorrect value D or inconsistent values to different honest parties in step 2(d)ii. Again, in both cases only share value correctness may be affected and real and simulated views remain indistinguishable.

Finally, in the simulation of the dot product gate, we note that interaction only occurs exactly as it would during the degree reconciliation phase of either type of multiplication gate. But this is precisely where (and only where) interaction occurs in those gates. Thus, indistinguishability between the real and simulated views of dot product gates follows from the preceding arguments for single round multiplication and linear communication multiplication.

Thus, the simulated view of Construction 1 is indistinguishable from real execution to $\mathcal{A}$. □

Proof of Lemma 4.5. Recall that to prove this, we need to show that extended linear-based protocols which are weakly private are secure up to additive attack (i.e. that the validity of this fact for (non-extended) linear based protocols, proved in [22], holds in light of these modifications). These details are treated in Appendix E. In order for this to be meaningful, we first need to show that the interactive building blocks we use in Construction 1 are linear as per Definition 4.1. To that end, note that each of the following protocols take setup based input in the form of any necessary encryption keys and randomness seeds. Moreover, the output of any of these building blocks is a linear combination of incoming messages, and thus by definition, constitutes a message of Type B in accordance with Definition 4.1. Otherwise, consider the following:

- share($[x]$): given that we instantiate secret sharing in this section with Shamir secret sharing, note that pre-computation involves sampling t random polynomial coefficients. Let P_i be the sharing party. P_i uses these to set vector $R = (r_1, r_2, \ldots, r_t)$. P_i then sends the matrix $J = (1, j, j^2, \ldots, j^t)_{1 \le j \le n}$ together with their main input value x to themselves as a Type A message as described in Definition 4.1. Next, P_i computes $[x]_j = R^T \cdot J[j]$ for $1 \le j \le n$. Note that these values each conform to the definition of a Type B message as described in Definition 4.1. Moreover, this shows that when holding a polynomial coefficient vector with element-0 as private input and all other elements as auxiliary input, parties can execute polynomial evaluation at any fixed point to obtain a Type B message. Finally, P_i sends $[x]_j$ to each party P_j (including themselves). Each party takes this vector to be their share and outputs it (which is trivially linear).
- reconstruct($[x]_J$): as described in Section 2, assumes a qualified subset J of parties (i.e. where $|J| \ge t+1$). A party $P_i \in J$ in the Shamir setting, cooperating to reconstruct $[x]$, proceeds as follows: P_i sends $[x]_i$ to each other $P_j \in J$. Given that all intervening computation from initial share() to this point has involved either local operations or one of the interactive sub-protocols discussed below (which we show to be linear), it follows inductively that $[x]_i$ is a linear combination of previous messages, and hence sending this constitutes a Type B message. P_i then computes (as all others in J do similarly) $x = \sum_{j \in J} [x]_j \cdot \alpha_J(j)$ where $\alpha_J(j) = \prod_{\substack{k \in J \\ k \ne j}} \frac{k}{k-j}$, and outputs x, which as required is a linear combination of their incoming messages.
- RandFld gate: Pre-computation for party P_i involves using setup based inputs to generate $\{\mathbb{F}\mathrm{PRG}(\mathsf{key}_Q)\}_{Q \in \mathcal{Q}, i \in Q}$. An (input-independent) linear combination of these values constitute share $[x]_i$, which is P_i's output.
- RandInt gate: Pre-computation for party P_i involves using setup based inputs to generate $\{\mathbb{Z}\mathrm{PRG}(\mathsf{key}_Q)\}_{Q \in \mathcal{Q}, i \in Q}$. An (input-independent) linear combination of these values constitute share $[x]_i$, which is P_i's output.
- Addition gate: On input $[x]$ and $[y]_i$, P_i sends to themselves as a Type A message, the shares $[x]_i$ and $[y]_i$ and output the linear combination $[z]_i = [x]_i + [y]_i$.
- Multiplication by a known field element: On input $[x]$ and $y \in \mathbb{F}$, P_i sends to themselves as a Type A message the share $[x]_i$. P_i then outputs linear combination $[z]_i = y \cdot [x]_i$.

- Multiplication gate (linear communication): As noted in Construction 1, pre-computation for party P_i involves using setup based inputs to generate $\langle 0\rangle_i$ and $[w]_i$. Next on input $[x]$ and $[y]_i$, P_i sends to themselves, as a Type A message, the value $\langle z\rangle_i = [x]_i \cdot [y]_i$ together with the constant share $[1]_i$. Next, P_i sends $\langle v\rangle_i = \langle z\rangle_i + \langle u\rangle_i$ to the king. Note that this is a linear combination of the previous message and auxiliary input, forming a Type B message. After the king (linearly) reconstructs v and broadcasts this value as a Type B message, P_i outputs $[z]_i = v - [w]_i$, which as required is a linear combination of their incoming messages.
- Multiplication gate (single round): For party P_i, let $J = \{j : i \bmod n \le j-1 \le (i+t) \bmod n\}$. As noted in Construction 1, pre-computation for party P_j involves using setup based inputs to generate $([d_i]_j)_{j\in J, j\neq i}$. Next on input $[x]$ and $[y]_i$, P_i sends to themselves, as a Type A message, the value $\langle z\rangle_i = [x]_i \cdot [y]_i$, which they define as $[d_i]_0$. P_j then reconstructs polynomial $f_{\langle z\rangle_i}$ from any $t+1$ $[d_i]_j$s (a linear combination of previous messages) and evaluates $f_{\langle z\rangle_i}$ for each $j \in J$, which as we showed above results in a Type B message. Each such $[d_i]_j$ is sent as such a message to each respective $P_j \in J$. Finally, P_i outputs $[z]_i = \sum_{j=1}^{n} \alpha_j [d_j]_i$, which as required is a linear combination of their incoming messages.
- Dot product gate: The only difference between dot product and either respective above multiplication is that the Type A message from each party to themselves includes $\langle z\rangle_i = \sum_{j=1}^{\ell} [x_j]_i \cdot [y_j]_i$ rather than $\langle z\rangle_i = [x]_i \cdot [y]_i$. This is still a valid function of primary inputs for such a message, and further analysis proceeds as in the respective multiplication gate.

□

B DELAYED VERIFICATION

Proof of Lemma 5.3. First notice that any gate which is downstream in G of some parties' input will be initially marked as needing verification during the first BFS (Line 3). Such gates are not reconsidered until Line 22, and hence this property holds at least until then. Moreover, any gate which is not downstream of any parties' input, but which is downstream of an attackable gate will be marked as needing verification in Line 16. Since this inclusive-or statement iteratively evaluates all incoming wires to a gate until either the needs-verification bit is set, or all incoming wires have been evaluated (per Line 18), then this property also holds until at least Line 22.

This shows that until Line 22, needing verification is a downstream monotone property within G whenever the value to be opened is a function of some parties' input, or is downstream of some attackable gate. Meanwhile, Line 22 is executed for each element of O_i, which contains an input-dependent addition gate followed by an open gate (together with input and output wires), as per Line 12.

Since the second BFS begins connected to all gates g where $g.R = 1$ and terminates propagation whenever it encounters a gate marked in the first BFS (in Line 20), then Line 22 only evaluates open gates with input values which are a function of some parties' input *and* have immediately preceding had added some random value which is not a function of any parties' input.

Notice that the last element in any tuple in O_i is an open gate, h where $h.I = 1$. Then the tuple contains all wires and gates associated with an additive random pad of some parties input. And if *either* input into the included addition gate is marked as not needing verification, *and* the range associated to h is $\mathbb{F}$, then Line 22 will mark the open gate as not needing verification. Note that by the invariance of marking input-dependent values as needing verification, the wire marked safe must be the random additive pad, and was marked safe precisely because it is well formed as per Definition 5.1 and meets the requirements of Definition 5.2.

Otherwise, all tuples in O_r contain only open gates h where $h.I = 0$, and these are reset to $h.V = 0$ in Line 24, as Definition 5.2 requires. □

C SUPPLEMENTAL INFORMATION FOR THE MAIN CONSTRUCTION

The ideal functionalities used in Construction 2 are given in Figures 1 and 2. A number of these functionalities (e.g., $\mathcal{F}_{\text{input}}$, $\mathcal{F}_{\text{mult}}$, $\mathcal{F}_{\text{randfld}}$, and $\mathcal{F}_{\text{zerocheck}}$) are modeled similar to their formulation in [9], and we introduce additional similarly formulated functionalities (e.g., $\mathcal{F}_{\text{dotprod}}$, $\mathcal{F}_{\text{randint}}$) to support the extended set of operations in this work. Note that $\mathcal{F}_{\text{mult}}$ and $\mathcal{F}_{\text{dotprod}}$ permit the adversary to introduce an additive error into the computation, while computation associated with other gates such as $\mathcal{F}_{\text{randfld}}$, $\mathcal{F}_{\text{randint}}$ and $\mathcal{F}_{\text{open}}$ is not attackable. We also note that in our instantiation of $\mathcal{F}_{\text{randint}}$, as given in Construction 1, $\nu = \binom{n}{t}$ and the element is drawn from the distribution corresponding to the sum of ν identical uniform variables over $\mathbb{Z}_{2^k}$, i.e., the Irwin-Hall or uniform sum distribution [28, 31].

Proof of Theorem 5.4. These two types of gates are attackable and the adversary can submit errors to each gate (as per Definition 5.1 and the corresponding ideal functionalities). If such an attack occurs, then since such gates compute either a multiplication of single shares or a dot product of shares, we can say $[z_i] = d_i + \sum_{j=1}^{h_i} ([x_j] \cdot [y_j])$, where $\{[x_j]\}_{j\in[1..h_i]}$ and $\{[y_j]\}_{j\in[1..h_i]}$ are the input shares, $[z_i]$ the output, and d_i the value the adversary adds to gate i. Note that there is a distinct vector size h_i for each gate i, and in particular, multiplication of single shares is a special case of dot product. That is, if gate i is multiplication of individual shares, then $h_i = 1$ and $\{x_j\}_{j\in[1..h_i]} = \{x_i\}$ and $\{y_j\}_{j\in[1..h]} = \{y_i\}$. In any event, only one d_i is added because all such protocols involve a single round of interaction. Similarly, on the randomized branch of the circuit, we have output share $[r \cdot z_i] = f_i + \sum_{j=1}^{h_i} [(r \cdot x_j + e_j)] \cdot [y_j]$. Here, the errors $\{e_j\}_{j\in[1..h_i]}$ are accumulated from previous calls to and attacks on attackable gates, and f_i is the error added by the adversary to the output of gate i on the randomized branch. Again, while there may be multiple upstream errors e_j, there is only a single attack f_i possible for gate i. We additionally denote additive attack error on the randomized input into the circuit by $r \cdot [v_i] + g_i$, and the additive attack error on the multiplication gate during the verification phase by $[r] \cdot [u_2] + k$. We use L to denote the number of inputs, and say i_0 is the first gate the adversary tampers with. In

Functionality $\mathcal{F}_{\text{input}}$

(1) $\mathcal{F}_{\text{input}}$ receives input $v \in \mathbb{F}$ from the party who owns that input and also receives from the adversary shares $[v]_C$ of the corrupted parties.
(2) $\mathcal{F}_{\text{input}}$ computes all shares of v as $([v]_1, \ldots, [v]_n) = \text{share}(v, [v]_C)$ and sends to each party P_i, $i \in [1, n]$, its output share $[v]_i$.

Functionality $\mathcal{F}_{\text{randfld}}$

(1) $\mathcal{F}_{\text{randfld}}$ receives shares $[s]_C$ from the adversary.
(2) $\mathcal{F}_{\text{randfld}}$ chooses a random element $w \in \mathbb{F}$, lets $[w]_j = [s]_j$ for each $p_j \in C$, and executes $([w]_1, \ldots, [w]_n) = \text{share}(w, [w]_C)$.
(3) $\mathcal{F}_{\text{randfld}}$ sends to each honest party P_i its share $[w]_i$.

Functionality $\mathcal{F}_{\text{randint}}(k)$

(1) $\mathcal{F}_{\text{randint}}$ receives shares $[s]_C$ from the adversary.
(2) $\mathcal{F}_{\text{randint}}$ chooses random $w \in \mathbb{Z}_{2^k \cdot v}$, where v and the distribution to draw w from depend on the protocol instantiation, lets $[w]_j = [s]_j$ for each $p_j \in C$, and executes $([w]_1, \ldots, [w]_n) = \text{share}(w, [w]_C)$.
(3) $\mathcal{F}_{\text{randint}}$ sends to each honest party p_j its share $[w]_j$.

Functionality $\mathcal{F}_{\text{mult}}$

(1) $\mathcal{F}_{\text{mult}}$ receives shares $[x]_H$ and $[y]_H$ from the honest parties and computes $x = \text{reconstruct}([x]_H)$ and $y = \text{reconstruct}([y]_H)$.
(2) $\mathcal{F}_{\text{mult}}$ computes the corrupt parties shares $[x]_C$ and $[y]_C$ and communicates them to the adversary.
(3) Upon receipt of value d and corrupt parties' shares $[z]_C$ from the adversary, $\mathcal{F}_{\text{mult}}$ defines $z = xy + d$ and computes $([z]_1, \ldots, [z]_n) = \text{share}(z, [z]_C)$.
(4) $\mathcal{F}_{\text{mult}}$ sends $[z]_i$ to each honest party P_i.

Functionality $\mathcal{F}_{\text{dotprod}}$

(1) $\mathcal{F}_{\text{mult}}$ receives shares $\{[x_i]_H\}_{i=1}^{\ell}$ and $\{[y_i]_H\}_{i=1}^{\ell}$ from the honest parties and computes $x_i = \text{reconstruct}([x_i]_H)$ and $y_i = \text{reconstruct}([y_i]_H)$ for $i \in [1, \ell]$.
(2) $\mathcal{F}_{\text{mult}}$ computes the corrupt parties shares $[x_i]_C$ and $[y_i]_C$ for $i \in [1, \ell]$ and communicates them to the adversary.
(3) Upon receipt of value d and corrupt parties' shares $[z]_C$ from the adversary, $\mathcal{F}_{\text{mult}}$ defines $z = d + \sum_{i=1}^{\ell} x_i y_i$ and computes $([z]_1, \ldots, [z]_n) = \text{share}(z, [z]_C)$.
(4) $\mathcal{F}_{\text{mult}}$ sends $[z]_i$ to each honest party P_i.

Figure 1: Ideal functionalities.

sum, we have:

$$[u_1] = \sum_{i=1}^{L} \alpha_i \cdot [r \cdot v_i + g_i] + \sum_{i=L+1}^{M} \alpha_i \cdot \left(f_i + \sum_{j=1}^{h_i} [(r \cdot x_j + e_j) \cdot y_j] \right)$$

$$[u_2] = \sum_{i=1}^{L} \alpha_i \cdot [v_i] + \sum_{i=L+1}^{M} \alpha_i \cdot \left(d_i + \sum_{j=1}^{h_i} [x_j \cdot y_j] \right)$$

Functionality $\mathcal{F}_{\text{open}}$

(1) $\mathcal{F}_{\text{open}}$ collects shares $[x]_i$ from each party.
(2) $\mathcal{F}_{\text{open}}$ calls reconstruct$([x]_J)$ on each unique set of $t+1$ shares $[x]_i$.
(3) If all reconstructions produce the same value x, then $\mathcal{F}_{\text{open}}$ broadcasts x to all parties. Otherwise, $\mathcal{F}_{\text{open}}$ signals abort.

Functionality $\mathcal{F}_{\text{output}}$

(1) $\mathcal{F}_{\text{output}}$ collects shares $[x]_i$ from each party.
(2) $\mathcal{F}_{\text{output}}$ calls reconstruct$([x]_J)$ on each unique set of $t+1$ shares $[x]_i$.
(3) If all reconstructions produce the same value x, then $\mathcal{F}_{\text{output}}$ sends x to party P_i. Otherwise $\mathcal{F}_{\text{output}}$ signals abort.

Functionality $\mathcal{F}_{\text{randfldpub}}$

(1) $\mathcal{F}_{\text{randfldpub}}$ receives shares $[s]_C$ from the adversary.
(2) $\mathcal{F}_{\text{randfld}}$ chooses a random element $w \in \mathbb{F}$ and sends it to all parties.

Functionality $\mathcal{F}_{\text{zerocheck}}$

(1) $\mathcal{F}_{\text{zerocheck}}$ receives shares $[v]_H$ from the honest parties and computes $v = \text{reconstruct}([v]_H)$.
(2) If $v = 0$, the adversary is given the opportunity to send accept or reject, which $\mathcal{F}_{\text{zerocheck}}$ conveys to the honest parties.
(3) If $v \neq 0$, $\mathcal{F}_{\text{zerocheck}}$ sends accept to the honest parties with probability $\frac{1}{|\mathbb{F}|}$ and reject with probability $1 - \frac{1}{|\mathbb{F}|}$.

Figure 2: Ideal functionalities (continued).

and further:

$$[T] = k + [u_1] - [r] \cdot [u_2]$$
$$= k + \sum_{i=1}^{L} \alpha_i \cdot [g_i] + \sum_{i=L+1}^{M} \alpha_i \cdot \left(f_i - r \cdot d_i + \sum_{j=1}^{h_i} [e_j \cdot y_j] \right) \quad (2)$$

- Case 1: the adversary first cheats during input randomization and does not cheat in the verification phase. Suppose g_{i_0} is the first error added in the inputs. Then if $[T] = 0$, we have

$$g_{i_0} \cdot \alpha_{i_0} = -\sum_{i=i_0+1}^{L} \alpha_i \cdot g_i - \sum_{i=L+1}^{M} \alpha_i \cdot \left(f_i - r \cdot d_i + \sum_{j=1}^{h_i} e_j \cdot y_j \right)$$

  Given the uniform distribution and independence of α_{i_0}, the probability of choosing the appropriate g_{i_0} in this case is $1/|\mathbb{F}|$.
- Case 2: the adversary first cheats in an attackable gate and does not cheat in the verification phase. Let the first errors be d_{i_0} and/or f_{i_0}. Note that in this case, $e_i = 0$ for all $i \leq i_0$ since this is the first error be introduced and thus no error is accumulated from previous gates. So $[T] = 0$ holds only if

$$\alpha_{i_0} \cdot (f_{i_0} - r \cdot d_{i_0}) = -\sum_{i=i_0+1}^{M} \alpha_i \cdot \left(f_i - r \cdot d_i + \sum_{j=1}^{h_i} e_j \cdot y_j \right) \quad (3)$$

  We first note that $\Pr[f_{i_0} = r \cdot d_{i_0}] = 1/|\mathbb{F}|$. This is because the value of r is uniformly random in $\mathbb{F}$ and unknown to the adversary during the computation phase. If instead $f_{i_0} \neq r \cdot d_{i_0}$ (which happens with probability $1 - 1/|\mathbb{F}|$), then the

probability that Equation 3 holds is similarly $1/|\mathbb{F}|$, since α_{i_0} is randomly chosen over $\mathbb{F}$ independent of all other values, and also unknown to the adversary at runtime. In sum, the probability that Equation 3 holds is bounded above by

$$\Pr[f_{i_0} = r \cdot d_{i_0}] + \Pr[(f_{i_0} \neq r \cdot d_{i_0}) \wedge ((3) \text{ holds})] = \frac{1}{|\mathbb{F}|} + \frac{1}{|\mathbb{F}|}\left(1 - \frac{1}{|\mathbb{F}|}\right) < \frac{2}{|\mathbb{F}|}$$

Note that this bound is proper since the event $\{f_{i_0} - r \cdot d_{i_0} = 0\}$ contains cases where Equation 3 does *not* hold.
Also note that although not provided a separate case, we can treat additive attacks on the randomizing step for RandFld and RandInt here as well. Note that this is similar to the attack on input randomization in case 1 (as these values conceptually are a different form of input), but can come after tampering with other gates, and in a non-predictable way with respect to timing and circuit topology. To do so, we note that if the initial randomization of a random share $[w_{i_0}]$ by multiplication with $[r]$ marks the first misbehavior by the adversary, then we can fix $y_{i_0} = 1$ on both branches of computation for this gate, and the only difference in Equation 3 above is that $d_{i_0} = 0$, while f_{i_0} represents the additive attack on the randomized branch. In this case, the probability of successful tampering is $1/|\mathbb{F}|$ by an identical argument to case 1.

- Case 3: the adversary cheats during multiplication in the verification phase. Then if $[T] = 0$, we have

$$k = -\sum_{i=1}^{L} \alpha_i \cdot g_i - \sum_{i=L+1}^{M} \alpha_i \cdot \left(f_i - r \cdot d_i + \sum_{j=1}^{h_i} e_j \cdot y_j\right)$$

In this case, the adversary has the knowledge of all randomness (i.e., $\alpha_1, \ldots, \alpha_M$) and all previous errors. However, $[r]$ is uniformly random in $\mathbb{F}$ and unknown to the adversary throughout the computation. Concretely, let $c_1 = -\sum_{i=1}^{L} \alpha_i \cdot g_i$, and $c_2 = -\sum_{i=L+1}^{M} \alpha_i \cdot \left(f_i + \sum_{j=1}^{h_i} e_j \cdot y_j\right)$, and let $c_3 = \sum_{i=L+1}^{M} \alpha_i \cdot d_i$. Then we have

$$k = rc_3 + c_1 + c_2$$

Thus, the probability of an adversary choosing the appropriate k to satisfy this equation while in possession of c_1, c_2, and c_3, is $1/|\mathbb{F}|$.

In all cases, the probability that adversary submits errors during a call to any $\mathcal{F}_{\text{mult}}$ and go undetected is at most $2/|\mathbb{F}|$ as required. □

Proof of Theorem 5.5. Given simulator $\mathcal{S}$, adversary $\mathcal{A}$ which controls the set of corrupt parties C, and trusted party $\mathcal{T}_{\mathcal{P}}$ computing function f, the protocols are simulated in the following manner:

(1) *Input sharing:* $\mathcal{S}$ collects from $\mathcal{A}$ each input v_i owned by a corrupt party, along with the corrupt parties' shares on these inputs. Then for each honest parties' input, $\mathcal{S}$ calculates $([v_i]_1, \ldots, [v_i]_n) = \text{share}(0, [v_i]_C)$ and sends to $\mathcal{A}$ the shares of corrupt parties $[v_i]_C$ on all honest parties' inputs. $\mathcal{S}$ stores all values.
(2) *Generation of randomization:* $\mathcal{S}$ collects from $\mathcal{A}$ the corrupt parties' shares, $[r]_C$. Then $\mathcal{S}$ generates a random $r \in \mathbb{F}$ and computes the shares $\text{share}(r, [r]_C) = ([r]_1, \ldots, [r]_n)$. $\mathcal{S}$ stores all values.
(3) *Randomization of inputs:* $\mathcal{S}$ simulates the ideal functionality $\mathcal{F}_{\text{mult}}$ in the multiplication of $[r]$ by input $[v_i]$ for $i = 1, ..., M$. For each input $[v_i]$, $\mathcal{S}$ gives shares $[v_i]_C$ and $[r]_C$ to $\mathcal{A}$. Next, $\mathcal{A}$ communicates to $\mathcal{S}$ the corrupt parties' shares of the product $[z_i]_C$ and the additive value g_i. $\mathcal{S}$ stores all the values it received from $\mathcal{A}$.
(4) *Function evaluation:* For each gate g of the following type, in the given topological order,
   (a) *Addition*: $\mathcal{S}$ computes the sum of the shares corresponding to each corrupt party $p_j \in C$ as in the protocol and stores the results.
   (b) *Multiplication by a known field element*: For given constant $c \in \mathbb{F}$, $\mathcal{S}$ multiplies the share of each corrupt party $p_j \in C$ by c and stores the result.
   (c) *Multiplication*: $\mathcal{S}$ simulates two invocations of $\mathcal{F}_{\text{mult}}$. $\mathcal{S}$ gives $\mathcal{A}$ the corrupt parties' shares on the input wires and consequently receives from $\mathcal{A}$ the corrupted parties output shares along with additive values d_k and f_k on the non-randomized and randomized circuits, respectively. $\mathcal{S}$ stores these values.
   (d) *Dot product*: $\mathcal{S}$ simulates two invocations of $\mathcal{F}_{\text{dotprod}}$. $\mathcal{S}$ gives $\mathcal{A}$ the corrupt parties' shares on the input wires and consequently receives from $\mathcal{A}$ the corrupted parties output shares along with additive values d_k and f_k on the non-randomized and randomized circuits, respectively. $\mathcal{S}$ stores these values.
   (e) RandFld *gate*: $\mathcal{S}$ collects the shares from $\mathcal{A}$ for g. $\mathcal{S}$ then samples a random value w according to B and the distribution for g. Finally, $\mathcal{S}$ calculates $([w_i]_1, \ldots, [w_i]_n) = \text{share}(w, [w_i]_C)$ and stores all values.
   (f) RandInt *gate*: $\mathcal{S}$ collects the shares from $\mathcal{A}$ for g. $\mathcal{S}$ then samples a random value w according to B and the distribution for g. Finally, $\mathcal{S}$ calculates $([w_i]_1, \ldots, [w_i]_n) = \text{share}(w, [w_i]_C)$ and stores all values.
   (g) *Opening*: If g is marked by Algorithm 1 as needing verification, then $\mathcal{S}$ proceeds to simulate Step 4h. Provided verification was successful, or if g is not marked as needing verification, $\mathcal{S}$ simulates $\mathcal{F}_{\text{open}}$ based on the values it has stored thus far, to receive value c. If $\mathcal{S}$ finds inconsistencies between values which would be opened, or if an efficient membership testing exists for B and $\mathcal{S}$ finds that $c \notin B$, then $\mathcal{S}$ signals abort to $\mathcal{T}_{\mathcal{P}}$ on behalf of all trusted parties. Otherwise, $\mathcal{S}$ announces the value of c to $\mathcal{A}$.
   (h) *Verification of evaluation:* $\mathcal{S}$ chooses random $\alpha_1, \ldots, \alpha_M$ and communicates these values to $\mathcal{A}$. Then $\mathcal{S}$ plays the role of the honest parties in simulating $\mathcal{F}_{\text{mult}}$ to $\mathcal{A}$ and receives from $\mathcal{A}$ the corrupted parties output shares along with additive values d_k and f_k on the non-randomized and randomized circuits, respectively. Next, $\mathcal{S}$ simulates $\mathcal{F}_{\text{zerocheck}}$. If any non-zero d_k, or f_k was provided to $\mathcal{F}_{\text{mult}}$ by $\mathcal{A}$ in the simulation, then $\mathcal{S}$ simulates $\mathcal{F}_{\text{zerocheck}}$ sending reject, and then all honest parties sending $\perp$, and signals abort to $\mathcal{T}_{\mathcal{P}}$ on behalf of all trusted parties. Otherwise, $\mathcal{S}$ proceeds to output reconstruction.

(5) *Output reconstruction:* $\mathcal{S}$ simulates one round of verification as per Step 4h. Provided no abort occurred, $\mathcal{S}$ sends to $\mathcal{T}_{\mathcal{P}}$ the input v_i it received from $\mathcal{A}$ in Step 1. $\mathcal{S}$ then receives from $\mathcal{T}_{\mathcal{P}}$ the output values for each corrupt party. Using these values, $\mathcal{S}$ simulates the honest parties participation in $\mathcal{F}_{\text{output}}$ for all parties. If in this stage $\mathcal{S}$ receives values from $\mathcal{A}$ which, based on the information it holds from the previous steps, and the output values received from $\mathcal{T}_{\mathcal{P}}$, $\mathcal{S}$ finds are incorrect, then for all honest P_j which would receive such value(s), $\mathcal{S}$ signals abort to $\mathcal{T}_{\mathcal{P}}$ on behalf of P_j, (and in such a case P_j does not receive output). Otherwise, $\mathcal{T}_{\mathcal{P}}$ will send output to all parties not signaling abort.

We will show that the view of the adversary is the same in the actual protocol execution as it is in the simulation, up to a probability of at most $3/|\mathbb{F}|$.

We first consider the case where some open (or output) gate g is marked by Algorithm 1 as needing verification. Note that in order for there to be any deviation between real and ideal execution, then at least one d_k or f_k is nonzero. Given this, $\mathcal{S}$ will, in simulation of $\mathcal{F}_{\text{zerocheck}}$, always reject. However, during execution of the actual protocol, $\mathcal{F}_{\text{zerocheck}}$ may accept if either

- $T = 0$, with probability at most $2/|\mathbb{F}|$, as in Theorem 5.4.
- $T \neq 0$ but $\mathcal{F}_{\text{zerocheck}}$ accepts, with probability $1/|\mathbb{F}|$, as per the definition of the functionality.

The probability with which this happens is bounded above by

$$\Pr[T = 0] + \Pr[(T \neq 0) \wedge (\mathcal{F}_{\text{zerocheck}} \text{ accepts})] < \frac{2}{|\mathbb{F}|} + \frac{1}{|\mathbb{F}|}\left(1 - \frac{2}{|\mathbb{F}|}\right) < \frac{3}{|\mathbb{F}|}$$

Note that even if successful tampering in one round of verification implied success in future rounds, which is not at all clear, this still would not increase the overall likelihood of successful tampering for $\mathcal{A}$ since verification rounds are atomic and sequential.

Then provided verification is successful with no abort being triggered, any tampering must solely be the result of $\mathcal{A}$ sending incorrect shares to $\mathcal{S}$ during simulation of $\mathcal{F}_{\text{open}}$. Because any $t + 1$ shares uniquely identifies all shares and the shared value and since this is the first tampering, the set of shares $\mathcal{S}$ computes on behalf of the honest parties will be sufficient for $\mathcal{S}$ to discover tampering with certainty, and in that case trigger abort. Note that this is identically distributed to what $\mathcal{A}$ would expect to see in real execution.

Next we consider any (open or output) g that is not marked as needing verification by Algorithm 1. Note that in this case, if $\mathcal{A}$ sends incorrect shares to $\mathcal{S}$ during $\mathcal{F}_{\text{open}}$, then as above they would trigger abort with certainty as would be expected in real computation. If not, then as shown in Lemma 5.3, this means that the value to be opened either

- is not a function of any parties main input
- or has been padded by an untamperable uniform random value drawn from $\mathbb{F}$ just prior to opening

In both above cases, note that $\mathcal{A}$ is aware of the function f being computed and the subset B assigned to the open gate.

In the former case, note that $\mathcal{S}$ will select some uniform random value from $\mathbb{F}$ in the randomness gate which was used (immediately prior) to additively pad the sensitive value. This will cause the opened value to also be uniformly distributed on $\mathbb{F}$, which is the expectation for such a gate in real execution.

In the latter case, notwithstanding incorrect shares, the only other avenue for skewed view distribution would be when testing membership in B, if one is being used. To that end, note that $\mathcal{S}$ sampled all randomized values used in calculating the value at g according to their associated subset B and distribution. And by design of the function f being computed, and the fact that local computation of randomness (as in Construction 2) is untamperable on an per-gate basis, then the value opened in simulation will match without error the subset and distribution prescribed for g, and $\mathcal{A}$'s view will identically distributed in simulation as it would be in real execution. Thus, open or output gates not marked as needing verification cannot skew the view distribution of $\mathcal{A}$.

In total, the statistical difference between the distributions of $\mathcal{A}$'s view in the real and simulated executions is negligible in κ. □

D EXTENSION TO SMALL FIELDS

As alluded to earlier, the construction for large fields can be viewed as a special case of the more general construction as seen in [9]. In the case of large (enough) fields, the probability of successful cheating is ensured to be negligible in κ by requiring that $|\mathbb{F}| \geq 3 \cdot (2^{\kappa})$. In principle, and extension to smaller fields can be achieved by requiring that $|\mathbb{F}| \cdot 2^{\delta} \geq 3 \cdot (2^{\kappa})$, where δ concurrent and independently randomized branches of computation are executed (and where $\delta = 1$ corresponds to the case of Construction 2). Then in the verification stage, if $\mathcal{F}_{\text{zerocheck}}$ rejects for *any* of the δ branches, then the protocol is aborted. The probability of successful adversarial behavior is bound on each branch in the same manner as above, and due to the independence of all randomly drawn values, the overall probability of such success is bounded by the product of these bounds, thus providing probability of successful cheating negligible in κ.

In order to realize this generalization, the necessary changes to Construction 2 include:

(1) In Step 2, shares $[r_i]$ are generated for $1 \leq i \leq \delta$.
(2) In Step 3 and every sub-step of Step 4, the parties compute and store tuples $([z], [r_1 z], ..., [r_\delta z])$.
(3) In Step 4h, the parties run δ branches of verification, including:
    (a) In Step 4(h)i, rather than obtaining public values α_i, β_i, and γ_i via calls to $\mathcal{F}_{\text{randfldpub}}$ and $\mathbb{F}$PRG, the parties obtain sets of shares $\{[\alpha_{i,1}], ..., [\alpha_{i,\delta}]\}_{i=1}^{M}$ via calls to $\mathcal{F}_{\text{randfld}}$.
    (b) Then, for each of the δ branches, Step 4(h)ii computation of linear combinations is achieved via $\mathcal{F}_{\text{dotprod}}$ in place of local multiplication, followed by a call to $\mathcal{F}_{\text{mult}}$ to calculate $[T]$ and executing $\mathcal{F}_{\text{zerocheck}}$ for that branch.

Regarding changes to Steps 4(h)i and 4(h)ii, the authors of [9] point out that when $\delta \geq 2$, generalizing the simulation strategy used in the proof of Theorem 5.5 will not suffice to prove the protocol secure. This is because if these random verification tokens were to be revealed, then any distinguisher with access to real execution inputs would be able to determine precisely for which values i it holds that $T_i \neq 0$. Since this simulator would be unable to do this, the tokens are instead kept as secret shares.

E EXTENDED LINEAR-BASED PROTOCOLS

Integral to the work of Genkin et al. [22] is the concept of a *linear protocol* (see Definition 4.1), where the output of each party is necessarily a linear combination of their incoming messages to the protocol, and the concept of a *linear-based protocol*, which compiles sub-protocols which are themselves *linear* into a larger framework under certain organizational constraints. We extend the definition of linear-based protocol to capture the additional protocols that we want to support to obtain Definition 4.2 (extended linear-based protocol), but the essence of the definition remains unchanged from that in [22]. In particular, instead of requiring that input-independent computation (e.g., triple or randomness generation for multiplication gates) takes place during the setup phase, we allow this pre-computation to be part of each linear protocol, as long as the input-independent pre-computation complies with the definition of the linear protocol itself. Excepting these changes, Definition 4.2 is a superset of linear-based protocol.

Genkin et al. define the output function a linear function (as per Definition 4.1), as follows:

Definition E.1 (Output function of a linear protocol). Let π be a linear protocol for computing a functionality f and let T be a set of parties. Let $\tilde{x}$ be a main input to π, let $\tilde{y}$ be an auxiliary input to π and let $m_{inp,T}$ be the messages of type a in Definition 4.1 sent by the parties in T to themselves during an honest execution of π on $(\tilde{x}, \tilde{y})$. In addition, let $m_{\overline{T},T}$ be the messages of type b sent by the parties in $\overline{T}$ to the parties in T during an honest execution of π. We say that a function out_T is the output function of T in π if for any main input $\tilde{x}$ and auxiliary input $\tilde{y}$ it holds that

$$\mathsf{out}_T(m_{inp,T}, \tilde{y}, m_{\overline{T},T}) = f_T(\tilde{x}, \tilde{y})$$

where f_T is the restriction of f to the outputs of the parties in T.

They also hold that the output function of a linear protocol as defined above is a linear function in the following sense: For any $m_1, y, m_2, m_1', y', m_2'$, it holds that

$$\mathsf{out}_T(m_1 + m_1', y + y', m_2 + m_2')$$
$$= \mathsf{out}_T(m_1, y, m_2) + \mathsf{out}_T(m_1', y', m_2')$$

Genkin et al. prove security of their linear-based construction by presenting a simulator (Construction 5.2 in [22]), which they show to be secure against a malicious adversary controlling exactly t servers (i.e. a maximal adversary in the honest-majority case, in their Lemma 5.3). In a separate proof they show (their Lemma 5.2) that more generally, any protocol secure against such a maximal adversary is also secure against an adversary controlling at most t servers. By combining these two proofs, they obtain a construction which is secure against any malicious adversary in the honest-majority case. This is their Theorem 5.2, and we state it here for reference:

Theorem E.2. *([22], Theorem 5.2) Let* Π *be a protocol computing a (possibly randomized) m-client circuit* $G : \mathbb{F}^{I_1} \times \cdots \times \mathbb{F}^{I_m} \to \mathbb{F}^{O_1}$ *using* $n = 2t + 1$ *servers that is linear-based with respect to some redundant and dense linear secret sharing scheme and is weakly-private against active adversaries controlling at most t servers and an arbitrary number of clients. Then,* Π *is a t-secure protocol for computing* $\tilde{f_C}$*, the additively corruptible version of G (as defined below).*

where the function $\tilde{f_C}$ is the one computed by a circuit which is subject to additive attack, defined as follows:

Definition E.3. ([22], Definition 5.1 - additively corruptible version of a circuit) Let $G : \mathbb{F}^{I_1} \times \cdots \times \mathbb{F}^{I_m} \to \mathbb{F}^{O_1}$ be an n-party circuit containing w wires. We define the additively corruptible version of G to be the n-party functionality $\tilde{f_C} : \mathbb{F}^{I_1} \times \cdots \times \mathbb{F}^{I_m} \to \mathbb{F}^{O_1}$ that takes additional input from the adversary which indicates an additive corruption for every wire of G. For all $(x, \mathbb{A})$, $\tilde{f_C}(x, \mathbb{A})$ outputs the result of the additively corrupted G as specified by the additive attack $\mathbb{A}$ ($\mathbb{A}$ is the simulators attack on G) when invoked on the inputs x.

Rather than state the simulator and proof in their entirety, we will treat only those sections which we modify or introduce (as per Definition 4.2) into the framework. As these changes only impact their Claim 5.4, that will be our focus. In particular, input sharing and output recovery remain unchanged, so we do not discuss these sections of Theorem 5.2. Let $\mathcal{S}$ be a simulator interacting with adversary $\mathcal{A}$. We use the following conventions:

(1) Probabilistic random variables appear in this script: $\mathbb{Z}$.
(2) Variables which represent specific instances taken from the support of random variables appear in normal math lower-case script, e.g. z.
(3) Variables which represent real-world computation are prefixed with $\mathcal{R}$, e.g. wire-value $\mathcal{R}z$.
(4) Variables which represent ideal-world computation are prefixed with $\mathcal{I}$, e.g. message $\mathcal{I}m$.
(5) Variables which represent the result of honest computation (i.e. as part of output from some ideal functionality π_{func}) are prefixed with $\mathcal{H}$, e.g. message $\mathcal{H}m$.

We now proceed to describe the simulator which will be featured in the proof of Lemma E.5 below, which is our extended version of Claim 5.4 in [22]:

(1) **Setup phase**. Note that π_{setup} gets no auxiliary inputs.
   (a) $\mathcal{S}$ receives from $\mathcal{A}$ the messages $\mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H}$ sent by the adversary to the honest servers during the execution of π_{setup}.
   (b) $\mathcal{S}$ simulates the behavior of the corrupt servers given their truncated view $\mathcal{I}u$ and computes the messages $\mathcal{H}m^{\text{sim},c}_{\mathcal{A}\to H}$ that should have been sent by the corrupt servers to the honest servers during the execution of π_{setup}. In addition, for every corrupt $P_i \in \mathcal{A}$ and every gate c, $\mathcal{S}$ computes the vector $\mathcal{H}g_i^{\text{sim},c}$ that is a part of the output of P_i after an honest execution of π_{setup}.
   (c) $\mathcal{S}$ computes $(\beta^c_H) \leftarrow \mathsf{out}_{H,\pi_{\text{setup}}}(0, \perp, \mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{c,\text{sim}}_{\mathcal{A}\to H})$.
(2) **Pre-computation**. Note that π_{precomp} gets no auxiliary inputs.
   (a) $\mathcal{S}$ receives from $\mathcal{A}$ the messages $(\mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H})_{1\le c\le |C|}$ sent by the adversary to the honest servers during the execution of π_{precomp} across all gates c.
   (b) $\mathcal{S}$ simulates the behavior of the corrupt servers given their truncated view $\mathcal{I}u$ and computes the messages

$\mathcal{H}m^{\text{sim},c}_{\mathcal{A}\to H}$ that should have been sent by the corrupt servers to the honest servers during the execution of π_{precomp}. In addition, for every corrupt $P_i \in \mathcal{A}$ and every gate c, $\mathcal{S}$ computes the vector $\mathcal{H}g^{\text{sim},c}_i$ that is a part of the output of P_i after an honest execution of π_{precomp}.

(c) $\mathcal{S}$ computes $(\gamma^c_H)_{1\le c\le|C|} \leftarrow \text{out}_{H,\pi_{\text{precomp}}}((\beta^c_H), \perp, \mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{c,\text{sim}}_{\mathcal{A}\to H})$. Note that it is implicit that $\mathcal{H}g^{\text{sim},c}_i = \gamma^c_H = \perp$ for non-interactive gates c.

(3) **Multiplication by a known field element**. Note that in this case no communication takes place. Thus, on main input $[\mathcal{I}x]^c_i$ and public constant value y, $\mathcal{S}$ computes $[\mathcal{I}z]^c_i = y \cdot [\mathcal{I}x]^c_i$ for $P_i \in \mathcal{A}$, and saves $\{[\mathcal{I}z]^c_i\}_{P_i\in\mathcal{A}}$ for later use.

(4) **Dot product gate**:

(a) $\mathcal{S}$ obtains from $\mathcal{A}$ the messages $\mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H}$ sent by the adversary to the honest servers during the execution of π_{dotprod}.

(b) $\mathcal{S}$ simulates the behavior of the corrupt servers on main inputs $\{[\mathcal{I}z_j]^a_i, [\mathcal{I}z_j]^b_i\}_{P_i\in\mathcal{A}, j\in[1,\ell]}$, auxiliary inputs $g^c_{\mathcal{A}}$, and from the truncated view $\mathcal{I}u$, incoming messages $\mathcal{H}m^{\text{sim},c}_{H\to\mathcal{A}}$ from the honest servers. $\mathcal{S}$ then computes the messages $\mathcal{H}m^{\text{sim},c}_{\mathcal{A}\to H}$ that should have been sent by the corrupt servers to the honest servers during the execution of π_{dotprod}. In addition, $\mathcal{S}$ computes the shares $\{[\mathcal{H}z]^{\text{sim},c}_i\}_{P_i\in\mathcal{A}}$ that represent the output of corrupt P_i after an honest execution of π_{dotprod}.

(c) $\mathcal{S}$ computes $\delta^c_H \leftarrow \text{out}_{H,\pi_{\text{mult}}}(0, \gamma^c_H, \mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{\text{sim},c}_{\mathcal{A}\to H})$ where γ^c_H are the part of γ_H corresponding to the gate c.

(d) Since SS is dense and redundant, $\mathcal{S}$ computes $\alpha^c \leftarrow \text{rec}(\delta^c_H, H)$, for every gate d connected to c, sets $A_{c,d} \leftarrow \alpha^c$, and computes the shares $\{\delta^c_i\}_{P_i\in\mathcal{A}}$ for the corrupt servers that are compatible with δ^c_H.

(e) $\mathcal{S}$ computes $[\mathcal{I}z]^c_i \leftarrow [\mathcal{H}z]^{\text{sim},c}_i + \delta^c_i$ for $P_i \in \mathcal{A}$, and saves $\{[\mathcal{I}z]^c_i\}_{P_i\in\mathcal{A}}$ for later use.

(5) RandFld **gate**:

(a) $\mathcal{S}$ receives from $\mathcal{A}$ the messages $\mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H}$ sent by the adversary to the honest servers during the execution of π_{randfld}.

(b) $\mathcal{S}$ simulates the behavior of the corrupt servers on auxiliary inputs $g^c_{\mathcal{A}}$, and from the truncated view $\mathcal{I}u$, incoming messages $\mathcal{H}m^{\text{sim},c}_{H\to\mathcal{A}}$ from the honest servers. $\mathcal{S}$ then computes the messages $\mathcal{H}m^{\text{sim},c}_{\mathcal{A}\to H}$ that should have been sent by the corrupt servers to the honest servers during the execution of π_{randfld}. In addition, $\mathcal{S}$ computes the shares $\{[\mathcal{H}z]^{\text{sim},c}_i\}_{P_i\in\mathcal{A}}$ that represent the output of corrupt P_i after an honest execution of π_{randfld}.

(c) $\mathcal{S}$ computes $\delta^c_H \leftarrow \text{out}_{H,\pi_{\text{randfld}}}(0, \gamma^c_H, \mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{\text{sim},c}_{\mathcal{A}\to H})$ where γ^c_H are the part of γ_H corresponding to the gate c.

(d) Since δ^c_H forms a valid sharing of some value, $\mathcal{S}$ computes $\alpha^c \leftarrow \text{rec}(\delta^c_H, H)$ and for every gate d connected to c, sets $A_{c,d} \leftarrow \alpha^c$. Since SS is redundant, $\mathcal{S}$ is able to compute the shares $\{\delta^c_i\}_{P_i\in\mathcal{A}}$ for the corrupt servers that are compatible with δ^c_H.

(e) $\mathcal{S}$ computes $[\mathcal{I}z]^c_i \leftarrow [\mathcal{H}z]^{\text{sim},c}_i + \delta^c_i$ for $P_i \in \mathcal{A}$, and saves $\{[\mathcal{I}z]^c_i\}_{P_i\in\mathcal{A}}$ for later use.

(6) RandInt **gate**:

(a) $\mathcal{S}$ receives from $\mathcal{A}$ the messages $\mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H}$ sent by the adversary to the honest servers during the execution of π_{randint}.

(b) $\mathcal{S}$ simulates the behavior of the corrupt servers on auxiliary inputs $g^c_{\mathcal{A}}$, and from the truncated view $\mathcal{I}u$, incoming messages $\mathcal{H}m^{\text{sim},c}_{H\to\mathcal{A}}$ from the honest servers. $\mathcal{S}$ then computes the messages $\mathcal{H}m^{\text{sim},c}_{\mathcal{A}\to H}$ that should have been sent by the corrupt servers to the honest servers during the execution of π_{randint}. In addition, $\mathcal{S}$ computes the shares $\{[\mathcal{H}z]^{\text{sim},c}_i\}_{P_i\in\mathcal{A}}$ that represent the output of corrupt P_i after an honest execution of π_{randint}.

(c) $\mathcal{S}$ computes $\delta^c_H \leftarrow \text{out}_{H,\pi_{\text{randint}}}(0, \gamma^c_H, \mathcal{I}m^{\text{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{\text{sim},c}_{\mathcal{A}\to H})$ where γ^c_H are the part of γ_H corresponding to the gate c.

(d) Since δ^c_H forms a valid sharing of some value, $\mathcal{S}$ computes $\alpha^c \leftarrow \text{rec}(\delta^c_H, H)$ and for every gate d connected to c, sets $A_{c,d} \leftarrow \alpha^c$. Since SS is redundant, $\mathcal{S}$ is able to compute the shares $\{\delta^c_i\}_{P_i\in\mathcal{A}}$ for the corrupt servers that are compatible with δ^c_H.

(e) $\mathcal{S}$ computes $[\mathcal{I}z]^c_i \leftarrow [\mathcal{H}z]^{\text{sim},c}_i + \delta^c_i$ for $P_i \in \mathcal{A}$, and saves $\{[\mathcal{I}z]^c_i\}_{P_i\in\mathcal{A}}$ for later use.

The following claim appears and is proved in [22], and we state it here for reference. It proves that any adversarial misbehavior in the setup phase induces an additive attack into the computation.

Lemma E.4. *([22], Claim 5.2) For any truncated view $\mathcal{R}u$ in the support of $\mathcal{RU}_{\mathcal{A}}$, it holds that $\mathcal{RG}^c_{H,\mathcal{R}u} \equiv \mathcal{HG}^c_{H,\mathcal{R}u} + \gamma^c_H$.*

The following lemma appears for (non-extended) linear-based protocols as Claim 5.4 in [22]. This claim asserts that the honest shares held by servers at the end of protocol execution and before reconstruction, correspond to a simulatable additive attack on the set of wires in the corresponding computation circuit. Afterwards, we prove that the claim still holds given the added functionality we have introduced under Definition 4.2. Recall that since the input sharing and output reconstruction is identical here, proof of this extended version of this claim suffices for proof of security of the extensions pursuant to Theorem 5.2 in [22]. Also note that in order to simplify notation, whenever the associated ideal functionality is understood (e.g. $\text{out}_{H,\pi_{\text{dotproduct}}}()$ and $\mathsf{IM}_{A,\pi_{\text{dotproduct}}}()$), it is omitted.

Lemma E.5. *For any truncated view $\mathcal{R}u$ in the support of $\mathcal{RU}_{\mathcal{A}}$, it holds that* $\left(\mathbb{Z}^1_u + \mathbb{A}^1_u, \ldots, \mathbb{Z}^{|C|}_u + \mathbb{A}^{|C|}_u\right) \equiv \left(\text{rec}_{H\cup\mathcal{A}}([\mathcal{RZ}]^1_H, [\mathcal{I}z]^1_{\mathcal{A}}), \ldots, \text{rec}_{H\cup\mathcal{A}}([\mathcal{RZ}]^{|C|}_H, [\mathcal{I}z]^{|C|}_{\mathcal{A}})\right)$.

Proof. Fix a truncated view $\mathcal{R}u$ from the support of $\mathcal{RU}_{\mathcal{A}}$. We now proceed to analyze the input sharing and circuit evaluation phases of $\mathcal{S}$ conditioned on $\mathcal{RU}_{\mathcal{A}} = \mathcal{R}u$. The proof is by induction on the structure of circuit C starting from the input gated and proceeding to the output gates.

Basis. If an input gate c belongs to a honest server then the claim is immediate since we have additive attack $\mathbb{A}^c = 0$. Similarly, if input gate c belongs to a corrupted server P_i, let x^c be the input of P_i to gate c. By construction of $\mathbb{A}^c$, we have that

$$\mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}v]^c_H, [\mathcal{I}v]^c_{\mathcal{A}}) \equiv x^c + \mathbb{A}^c_u \equiv \mathbb{Z}^c_u + \mathbb{A}^c_u$$

For a RandFld or RandInt gate c, the intuition is that $\mathcal{A}$ can only tamper in the pre-computation phase, which necessarily constitutes an additive attack. More concretely, (using RandFld as a basis, with RandInt similar):

$$\begin{aligned}
&\mathsf{rec}_H\left([\mathcal{R}z]^c_H, [\mathcal{I}z]^c_{\mathcal{A}}\right) = \mathsf{rec}_H\left(\mathsf{out}_H\left(0, \mathcal{R}g_{H,\mathcal{R}u}, \mathcal{I}m^{\mathsf{rcvd},c}_{\mathcal{A}\to H}\right)\right)\\
&= \mathsf{rec}_H\left(\mathsf{out}_H\left(0, \mathcal{R}g_{H,\mathcal{R}u} + \gamma^c_H - \gamma^c_H,\right.\right.\\
&\qquad\left.\left.\mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H} + \mathcal{I}m^{\mathsf{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\right)\right)\\
&= \mathsf{rec}_H\left(\mathsf{out}_H\left(0, \mathcal{R}g_{H,\mathcal{R}u} - \gamma^c_H, \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\right)\right)\\
&\qquad + \mathsf{rec}_H\left(\mathsf{out}_H\left(0, \gamma^c_H, \mathcal{I}m^{\mathsf{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\right)\right) \qquad (4)\\
&= \mathsf{rec}_H\left(\mathsf{out}_H\left(0, \mathcal{H}g_{H,\mathcal{R}u}, \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\right)\right) + \delta^c_H \qquad (5)\\
&\equiv \mathbb{Z}^c_u + \mathbb{A}^c_u
\end{aligned}$$

where δ^c are the value(s) computed in simulator Steps 5c and 5d, and δ^c_H and δ^c_C the restrictions of δ^c to, respectively, the honest and corrupt servers. Note that the transition in Equation 4 follows from the linearity of out, π_{randfld}, and $\mathcal{SS}$, and Equation 5 follows from Lemma E.4 and Step 5c of the simulator. Denote by $[\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}}$ the shares obtained during simulator Step 5b. Notice that these shares are completely determined by the truncated view $\mathcal{R}u$ and $\mathcal{A}$, and that by construction of $[\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}}$ we have

$$\mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{R}z]^c_H - \delta^c_H, [\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}}\right) = \mathsf{rec}_H\left([\mathcal{R}z]^c_H - \delta^c_H\right) \equiv \mathbb{Z}^c_u.$$

Since we also have that $[\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}} + \delta^c_{\mathcal{A}} = [\mathcal{I}z]^c_{\mathcal{A}}$, then by the linearity of $\mathcal{SS}$ we obtain

$$\begin{aligned}
&\mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{R}z]^c_H, [\mathcal{I}z]^c_{\mathcal{A}}\right)\\
&\quad= \mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{R}z]^c_H, [\mathcal{I}z]^c_{\mathcal{A}}\right) - \mathsf{rec}_{H\cup\mathcal{A}}\left(\delta^c\right) + \mathsf{rec}_{H\cup\mathcal{A}}\left(\delta^c\right)\\
&\quad= \mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{R}z]^c_H - \delta^c_H, [\mathcal{I}z]^c_{\mathcal{A}} - \delta^c_{\mathcal{A}}\right) + \mathsf{rec}_{H\cup\mathcal{A}}\left(\delta^c\right)\\
&\quad= \mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{R}z]^c_H - \delta^c_H, [\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}}\right) + \mathsf{rec}_{H\cup\mathcal{A}}\left(\delta^c\right)\\
&\quad\equiv \mathbb{Z}^c_u + \mathbb{A}^c_u
\end{aligned}$$

Induction hypothesis. Assume that for any $1 \le c \le |C|$ it holds that $(\mathbb{Z}^1_u + \mathbb{A}^1_u, \ldots, \mathbb{Z}^{c-1}_u + \mathbb{A}^{c-1}_u) \equiv (\mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}\mathbb{Z}]^1_H, [\mathcal{I}z]^1_{\mathcal{A}}), \ldots, \mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}\mathbb{Z}]^{c-1}_H, [\mathcal{I}z]^{c-1}_{\mathcal{A}}))$.

Induction step. Let c be a gate inside C with inputs a and b such that $c > a$ and $c > b$. Assume without loss of generality that $b > a$ and fix values $([\mathcal{R}z]^1_H, \ldots, [\mathcal{R}z]^b_H)$ from the support of $([\mathcal{R}\mathbb{Z}]^1_H, \ldots, [\mathcal{R}\mathbb{Z}]^b_H)$, and $(z^1, \ldots, z^b)$ from the support of $(\mathcal{Z}^1, \ldots, \mathcal{Z}^b)$. We now claim that conditioned on the above selection it holds that $\mathbb{Z}^c_u + \mathbb{A}^c_u \equiv \mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}\mathbb{Z}]^c_H, [\mathcal{I}z]^c_{\mathcal{A}})$.

From the induction hypothesis we have that,

$$z^a_u + \mathbb{A}^a_u \equiv \mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}z]^a_H, [\mathcal{I}z]^a_{\mathcal{A}})$$

and

$$z^b_u + \mathbb{A}^b_u \equiv \mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}z]^b_H, [\mathcal{I}z]^b_{\mathcal{A}})$$

Notice that the random variable $\mathbb{Z}^c_u$ conditioned on the above selection of $([\mathcal{R}z]^1_H, \ldots, [\mathcal{R}z]^b_H)$ and $(z^1_H, \ldots, z^b_H)$ is uniquely determined by the fixed $z^a_u + \mathbb{A}^a_u$ and $z^b_u + \mathbb{A}^b_u$.

Handling dot product gates. If c is a dot product gate, then notice that by the definition of $\tilde{f}_C$, by the linearity of $\mathcal{SS}$, and by the induction hypothesis it holds that

$$\begin{aligned}
\mathbb{Z}^c_u &\equiv (z^a_u + \mathbb{A}^a_u)\cdot(z^b_u + \mathbb{A}^b_u) \equiv \sum_{i=1}^{\ell}(z^{a_i}_u + \mathbb{A}^{a_i}_u)(z^{b_i}_u + \mathbb{A}^{b_i}_u)\\
&\equiv \sum_{i=1}^{\ell}(\mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}z]^a_H, [\mathcal{I}z]^a_{\mathcal{A}}))(\mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}z]^b_H, [\mathcal{I}z]^b_{\mathcal{A}}))
\end{aligned}$$

Next, denote by $\mathbb{IM}^{\mathsf{rcvd},c}_{\mathcal{A}\to H}$ the messages obtained by $\mathcal{S}$ in simulator Step 4a and denote by $\mathcal{RM}^c_{\mathcal{A}\to H}$ the messages obtained by the honest servers corresponding to the output of gate c during a real execution of the protocol. Since we have that $\mathbb{IU}_{\mathcal{A}} \equiv \mathcal{RU}_{\mathcal{A}}$, it holds that $\left(\mathbb{IU}_{\mathcal{A}}, \mathbb{IM}^{\mathsf{rcvd},c}_{\mathcal{A}\to H}\right) \equiv \left(\mathcal{RU}_{\mathcal{A}}, \mathcal{RM}^c_{\mathcal{A}\to H}\right)$.

Notice that the truncated view $\mathcal{R}u$ and $\mathcal{A}$ completely determine both $\mathbb{IM}^{\mathsf{rcvd},c}_{\mathcal{A}\to H}$ and $\mathcal{RM}^c_{\mathcal{A}\to H}$ as well as the messages $\mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}$ obtained by the simulator in simulator Step 4b. Thus, denote by $(\mathcal{R}\cup\mathcal{I})m^{\mathsf{rcvd},c}_{\mathcal{A}\to H}$ the value of both $\mathbb{IM}^{\mathsf{rcvd},c}_{\mathcal{A}\to H}$ and $\mathcal{RM}^c_{\mathcal{A}\to H}$ as determined by $\mathcal{R}u$.

For any set of servers T denote by $\mathsf{IM}_{T,\pi_{\mathsf{dotprod}}}(x_T)$ the nondeterministic functionality that computes the values of the input-dependent messages that the servers in T send to all the servers given their deterministic inputs x_T and using fresh randomness.

By Lemma E.4 we have that $\mathcal{RG}^c_{H,\mathcal{R}u} - \gamma^c_H \equiv \mathcal{HG}^c_{H,\mathcal{R}u}$ Thus it holds that

$$\begin{aligned}
&\mathsf{rec}_H\left(\mathsf{out}_H\left(\mathsf{IM}_{\mathcal{A}}\left(\textstyle\sum_{i=1}^{\ell}[\mathcal{R}z]^a_H, \sum_{i=1}^{\ell}[\mathcal{R}z]^b_H\right),\right.\right.\\
&\qquad\left.\left.\mathcal{RG}^c_{H,\mathcal{R}u} - \gamma^c_H, \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\right)\right) \qquad (6)\\
&\equiv \mathsf{rec}_H\left(\mathsf{out}_H\left(\mathsf{IM}_{\mathcal{A}}\left(\textstyle\sum_{i=1}^{\ell}[\mathcal{R}z]^a_H, \sum_{i=1}^{\ell}[\mathcal{R}z]^b_H\right),\right.\right.\\
&\qquad\left.\left.\mathcal{HG}^c_{H,\mathcal{R}u}, \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\right)\right)\\
&= \sum_{i=1}^{\ell}\left(\mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{R}z]^a_H, [\mathcal{H}z]^a_{\mathcal{A}}\right)\right)(\mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{R}z]^b_H, [\mathcal{H}z]^b_{\mathcal{A}}))\\
&\equiv \sum_{i=1}^{\ell}(z^a_u + \mathbb{A}^a_u)(z^b_u + \mathbb{A}^b_u) = \mathbb{Z}^c_u
\end{aligned}$$

Second, notice that

$$\begin{aligned}
\mathsf{rec}_H\left([\mathcal{RZ}]^c_H\right) &= \mathsf{rec}_H\Big(\mathsf{out}_H\Big(\mathsf{IM}_{\mathcal{A}}\Big(\textstyle\sum_{i=1}^{\ell}[\mathcal{R}z]^a_H, \sum_{i=1}^{\ell}[\mathcal{R}z]^b_H\Big),\\
&\qquad\qquad \mathcal{R}g_{H,\mathcal{R}u}, (\mathcal{R}\cup\mathcal{I})m^{\mathsf{rcvd},c}_{\mathcal{A}\to H}\Big)\Big)\\
&= \mathsf{rec}_H\Big(\mathsf{out}_H\Big(\mathsf{IM}_{\mathcal{A}}\Big(\textstyle\sum_{i=1}^{\ell}[\mathcal{R}z]^a_H, \sum_{i=1}^{\ell}[\mathcal{R}z]^b_H\Big),\\
&\qquad\qquad \mathcal{R}g_{H,\mathcal{R}u} + \gamma^c_H - \gamma^c_H, \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\\
&\qquad\qquad\qquad + \ (\mathcal{R}\cup\mathcal{I})m^{\mathsf{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\Big)\Big)\\
&= \mathsf{rec}_H\Big(\mathsf{out}_H\Big(\mathsf{IM}_{\mathcal{A}}\Big(\textstyle\sum_{i=1}^{\ell}[\mathcal{R}z]^a_H, \sum_{i=1}^{\ell}[\mathcal{R}z]^b_H\Big),\\
&\qquad\qquad \mathcal{R}g_{H,\mathcal{R}u} - \gamma^c_H, \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\Big)\Big) \qquad (7)\\
&\quad + \mathsf{rec}_H\Big(\mathsf{out}_H\Big(0, \gamma^c_H, (\mathcal{R}\cup\mathcal{I})m^{\mathsf{rcvd},c}_{\mathcal{A}\to H} - \mathcal{H}m^{\mathsf{sim},c}_{\mathcal{A}\to H}\Big)\Big)\\
&\equiv \mathbb{Z}^c_u + \mathbb{A}^c_u \qquad (8)
\end{aligned}$$

where the transition in Equation 7 follows the linearity of out, π_{dotprod}, and $\mathcal{SS}$, and Equation 8 follows from Equation 6 and the construction of $\mathbb{A}$. Thus, we have proved that $\mathsf{rec}_H\left([\mathcal{RZ}]^c_H\right) = \mathbb{Z}^c_u + \mathbb{A}^c_u$. Denote by $[\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}}$ the shares obtained during simulator Step 4b. Notice that these shares are completely determined by the truncated view u and $\mathcal{A}$. In addition, let δ^c be the values computed in simulator Steps 4c and 4d, and δ^c_H and δ^c_C the restrictions of δ^c to, respectively, the honest and corrupt servers. Notice that these values are also completely determined by $\mathcal{R}u$ and $\mathcal{A}$. Notice that by construction we have that $\mathsf{rec}_{H\cup\mathcal{A}}(\delta^c) = \mathbb{A}^c_u$, and thus by the linearity of $\mathcal{SS}$ it holds that

$$\mathsf{rec}_H\left([\mathcal{RZ}]^c_H - \delta^c_H\right) \equiv \mathbb{Z}^c_u$$

Next, notice that since $\mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{R}z]^a_H, [\mathcal{I}z]^a_{\mathcal{A}}\right) \neq \perp$ and $\mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{R}z]^b_H, [\mathcal{I}z]^b_{\mathcal{A}}\right) \neq \perp$, then by the construction of $[\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}}$ we have that

$$\mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{RZ}]^c_H - \delta^c_H, [\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}}\right) = \mathsf{rec}_H\left([\mathcal{RZ}]^c_H - \delta^c_H\right) \equiv \mathbb{Z}^c_u$$

Finally, since by construction we have that $[\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}} + \delta^c_{\mathcal{A}} = [\mathcal{I}z]^c_{\mathcal{A}}$, then by the linearity of $\mathcal{SS}$ we obtain

$$\begin{aligned}
&\mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{RZ}]^c_H, [\mathcal{I}z]^c_{\mathcal{A}}\right)\\
&\quad = \mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{RZ}]^c_H, [\mathcal{I}z]^c_{\mathcal{A}}\right) - \mathsf{rec}_{H\cup\mathcal{A}}\left(\delta^c\right) + \mathsf{rec}_{H\cup\mathcal{A}}\left(\delta^c\right)\\
&\quad = \mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{RZ}]^c_H - \delta^c_H, [\mathcal{I}z]^c_{\mathcal{A}} - \delta^c_{\mathcal{A}}\right) + \mathsf{rec}_{H\cup\mathcal{A}}\left(\delta^c\right)\\
&\quad = \mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{RZ}]^c_H - \delta^c_H, [\mathcal{H}z]^{\mathsf{sim},c}_{\mathcal{A}}\right) + \mathsf{rec}_{H\cup\mathcal{A}}\left(\delta^c\right)\\
&\quad \equiv \mathbb{Z}^c_u + \mathbb{A}^c_u
\end{aligned}$$

Thus we have proved that

$$\mathsf{rec}_{H\cup\mathcal{A}}\left([\mathcal{RZ}]^c_H, [\mathcal{I}z]^c_{\mathcal{A}}\right) = \mathsf{rec}_H\left([\mathcal{RZ}]^c_H\right) = \mathbb{Z}^c_u + \mathbb{A}^c_u$$

Therefore, for any $1 \le c \le |C|$, it holds that

$$\begin{aligned}
&\left(\mathbb{Z}^1_u + \mathbb{A}^1_u, ..., \mathbb{Z}^c_u + \mathbb{A}^c_u\right)\\
&\equiv \left(\mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{RZ}]^1_H, [\mathcal{I}z]^1_{\mathcal{A}}), ..., \mathsf{rec}_{H\cup\mathcal{A}}([\mathcal{RZ}]^c_H, [\mathcal{I}z]^c_{\mathcal{A}})\right)
\end{aligned}$$

and the claim follows. □