



# When Trying to Catch Cheaters Breaks the MPC: Breaking and Fixing Delayed Consistency Checks in Trident, Fantastic Four, SWIFT, and Quad

Andreas Brüggemann<sup>(✉)</sup> and Thomas Schneider<sup>(ID)</sup>

Technical University of Darmstadt, Darmstadt, Germany  
{brueggemann,schneider}@encrypto.cs.tu-darmstadt.de

**Abstract.** Actively secure multi-party computation in the honest-majority setting often relies on multiple parties computing the same message to be sent. This additional redundancy allows to detect when a party deviates from the protocol. Many works utilize this for efficient protocol design, with some protocols delaying and batching consistency checks to further boost efficiency. In this paper, we show multiple cases where such batched consistency checks render the protocols insecure. Our concrete attacks derive additional knowledge from the batched consistency checks, reconstructing values on intermediate wires. Specifically, we show concrete attacks on Trident (NDSS’20), Fantastic Four (USENIX Security’21) including its implementation in the popular MP-SPDZ framework (CCS’20), and Quad (PoPETS’25). Furthermore, we find how an imprecise specification of SWIFT (USENIX Security’21) can enable a similar attack and reveal a gap in their security proof. Finally, we propose a fix for all protocols with a small performance overhead. Our provably secure fix uses a generic, joint consistency check that replaces the former, insecure consistency checks.

## 1 Introduction

Secure multi-party computation (MPC) enables multiple parties to compute a function on their joint private inputs without any need to disclose their inputs to each other. Especially MPC with a low number of parties and an honest majority where the adversary can only corrupt less than half of the parties is receiving lots of attention due to the availability of highly efficient protocols, e.g., [1, 3, 10, 21, 30]. Motivating applications are secure computations of large functionalities such as privacy-preserving machine learning [10, 30] or analysis of large graphs [2, 26]. In the active/malicious security model, where the adversary can arbitrarily change the behavior of corrupted parties, an honest majority was shown especially useful, exploiting redundancy of intermediate data to detect deviations from the protocol [6, 7, 12, 14, 17, 21, 28].

In this work, we investigate the family of protocols from [12, 14, 21, 28], designed to be secure against a single malicious party. They detect protocol

deviations by having for each message sent by some party  $P_A$  a second party  $P_B$  who can compute and provide the same message, hence introducing redundancy. The party that receives the message from  $P_A$  should get the same information from  $P_B$ , and in case of any inconsistency between the received messages, securely aborts the protocol execution. To avoid overhead from sending redundant information, the redundant material sent by  $P_B$  is delayed during an optimistic protocol execution, where only in the end, before revealing any outputs, redundancy information is transmitted in a batched, compressed way, resulting in zero amortized overhead. Intuitively, this will still detect any inconsistency introduced by a cheating party. In this work, we observe that without an immediate consistency check, a cheater can provide incorrect data to an honest party  $P_A$  without immediately being detected, “poisoning” the internal state of  $P_A$ . Now, an operation using this poisoned state where  $P_A$  sends a message to the cheater eventually requires another honest party  $P_B$  to provide the same, redundant information to the cheater to ensure consistency. This may result in an additional inconsistency, given that  $P_A$  computes its message on poisoned data so that it can deviate from what  $P_B$  sends. The cheater then tries to extract additional information, using the difference between the messages received from  $P_A, P_B$ . Such an attack is enabled only by the delayed consistency check. Otherwise, a corrupted party would always receive consistent data from the immediate consistency check, given that the other parties are honest. On the other side, should a party receive inconsistent data during a check, one of the providers would be malicious and, thus, the receiver learns more information, but must be honest, preserving the protocol’s security. Yet, the protocols’ outstanding performance relies on not having to check for inconsistency immediately after each step.

## 1.1 Our Contributions

Our contributions are summarized in the following, and an overview of our attacks on the protocols in [12, 14, 21, 28] is given in Table 1.

**Vulnerability of Delayed Consistency Checks:** We identify and point out the aforementioned generic attack pattern on delayed consistency checks, where a cheater poisons the state of an honest party without immediate detection, using this and later consistency checks to extract private information. Doing that, we observe a dangerous trend where the delayed, batched consistency check is textually described in the respective papers while not being properly included in the formal protocol specification. The original proofs implicitly assume the unoptimized, formal protocol specification, proving secure only an intermediate, simplified, and less efficient version of the protocol. It appears that this discrepancy and the resulting vulnerability have then been carried along in research for more improved protocols without being noticed, given that [14, 21] even cite [28] as the basis for their consistency checks. The reuse of the vulnerable construction appears to be ongoing, given the newness of [21] and the same idea appearing in more works such as [7, 29] while not always opening an attack surface.

**Table 1.** Number of parties and corrupted parties, exact security model (within active security), amortized cost per multiplication (offline + online), vulnerability to our attack, and global communication cost of our proposed fix w.r.t. the evaluated circuit.

Protocol	#Parties	#Corr.	Security	cost/mult.	Insecure?	Fix overhead
Trident [12]	(§3) 3 + 1	1	Fair	3 + 3	Yes	$\mathcal{O}(1)$
Fantastic Four [14](§4)	4	1	Robust	0 + 6	Yes	$\mathcal{O}(1)/\mathcal{O}(d)$ <sup>a</sup>
SWIFT [28]	(§5) 3	1	Robust	3 + 3	Depends <sup>b</sup>	- <sup>c</sup>
Quad [21]	(§6) 3 + 1	1	Fair	2 + 3	Yes	$\mathcal{O}(1)$

<sup>a</sup>  $d$ : circuit depth;  $\mathcal{O}(1)$  for fair protocol variant;  $\mathcal{O}(d)$  for robust variant

<sup>b</sup> Protocol underspecified, possible protocol within the specification is insecure

<sup>c</sup> Protocol appears fixable by appropriate, unambiguous specification

**Concrete Attacks:** Using our generic attack, we discover explicit attacks on the protocols of Trident [12], Fantastic Four [14], SWIFT [28], and Quad [21]. For Trident, Fantastic Four, and Quad, flaws in the consistency check enable testing for or even extracting values on input wires to multiplication/AND-gates in arithmetic/binary circuits, especially for gates on the last layer of a circuit. In SWIFT, another orthogonal optimization appears to almost inadvertently thwart our attack, still leaving a gap in the security proof, as the optimization is not consciously used for security reasons. Still, we find that SWIFT’s under-specified input phase can still enable the extraction of values on some wires.

**Efficient Fixes:** We provide a generic construction that mitigates our attack by outsourcing the complete consistency check to a small MPC protocol execution. Prior consistency checks consist of multiple partial checks for each pair of sender and receiver, and the partial check where a cheater initially provides incorrect data will always fail. By batching all individual checks in a single protocol that outputs only a single bit, indicating if *any* inconsistency has been detected (but not which one), a cheating attempt will always lead to output 0, leaking nothing. We prove that our fix is secure, i.e., that it maintains soundness of the verification and does not leak any unintended data. It comes at zero amortized and a low concrete overhead with low, constant rounds. We instantiate and use it to fix Trident [12], Fantastic Four [14] with fairness, and Quad [21]. Regarding the robust variant of Fantastic Four, we propose a fix that comes at an overhead linear in the circuit depth, and we fix SWIFT [28] by a less ambiguous specification.

**Impact on MPC Implementations:** Several MPC implementations were vulnerable to our attacks. Fantastic Four [14] is available in MP-SPDZ [24], the most popular code framework for MPC, and directly inherits the flaw of the insecure protocol specification. Furthermore, the recent framework HPMPC [20] published together with Quad [21] provides not only an implementation of Quad, but also implements Fantastic Four, inheriting the security flaws of two protocols. Finally, the oblivious analytics system ORQ [4] uses a vulnerable implementation of Fantastic Four as one of its protocol back ends.

**Responsible Disclosure:** We have notified the maintainers of the affected implementations MP-SPDZ [24], HPMPC [20], and ORQ [4]. Together, we have coordinated that the issues were addressed before our attack was publicly disclosed. Simultaneously, we have contacted all authors of the papers proposing vulnerable protocols considered in this work [12, 14, 21, 28]. For more information, see §7.

## 1.2 Related Work

Our attack essentially introduces an error in one path of the computation and uses another correctly executed path, intended to normally provide redundancy, to generate an inconsistency and extract secrets from that. The concept of introducing errors in some computation paths is related to fault attacks, and a prominent example is [5]. There, random or purposefully induced faults are used to extract secret keys in signature and identification schemes, given that no check for faults is done before faulty data is distributed.

In the space of MPC, our attack is conceptually similar to the “double-dipping attack” of [15, 19]. This attack uses redundancy in DN-style [16] protocols for corruption thresholds where the double-degree polynomials are not full-threshold and, hence, not all shares are required for reconstruction. A malicious party exploits that by providing incorrect data to one party during a multiplication. In a subsequent multiplication, this party then provides incorrect data, inconsistent with the shares provided by the other parties. Like our attack, this is enabled as verification is delayed, allowing incorrect data to be used by honest parties before detection, while there also is redundancy. The pattern of attacking two consecutive multiplications is furthermore equal to what we will use for our attack. Our work differs in that the redundancy that is crucial for extracting secrets is introduced by the verification of the protocols we attack, whereas in the double-dipping attack, it is more centrally a part of the main protocol execution without verification. This difference also necessitates a fix to the verification in our case, whereas [15, 19] deploy their fix in the protocol’s main execution part. For a more detailed comparison to our work, we refer to the full version [8, App. E].

While our attack uses redundancy provided by a verification step of the targeted protocols, we emphasize that it is not a selective failure attack [25], where a malicious party can make the protocol conditionally abort, depending on other parties’ private information. In our case, a party cheating is always detected. This detection consists of several partial detection steps, where individual ones may still fail depending on private inputs. Still, we are able to extract significantly more information than this single bit (per detection step).

Our fix (§3.5) generalizes the idea of [22] who compare two hashes with an MPC protocol in the two-party setting to multiple hashes and parties.

**Outline.** After the preliminaries (§2), we provide a detailed explanation of our attack on Trident [12] in §3, including a generic construction to fix the underlying issues inexpensively. After this introduction of the attack’s concept and fix,

we translate these to Fantastic Four [14] (§4) while also discussing additional challenges to reach robustness. In §5, we discover that SWIFT’s [28] structure almost thwarts our generic attack, but document a gap in its security proof and show possible alternative attacks. Finally, in §6 we briefly remark on how our attack and fix translate to Quad [21] with details in the full version [8, App. D].

## 2 Preliminaries

**Notation.** In this work, we mainly consider parties  $P_1, \dots, P_n$ . In some cases, we may change the numbering to be consistent with the respective original protocol later. In some protocols, there is one party  $P_h$  acting as a helper that is absent from most or all of the protocol’s online phase. In this case, we may explicitly define that  $P_h$  not to be among  $P_1, \dots, P_n$  for ease of notation. When referring to parties, we may write, e.g.,  $P_{i+1}$  for  $1 \leq i \leq n$  to refer to the next party after  $P_i$  in a circular way where  $P_{n+1}$  refers to  $P_1$  again. Likewise, we may write  $P_{i-1}$  with  $P_{1-1}$  referring to  $P_n$ .  $P_h$  is not considered in this notation.

Computation is over a commutative and finite ring  $\mathcal{R}$  which usually is either  $\mathbb{Z}_{2^\ell}$  for  $\ell$ -bit integers with  $\ell \in \mathbb{N}$ , or a finite field  $\mathbb{F}_{p^k}$  for prime  $p$  and  $k \in \mathbb{N}$ .<sup>1</sup> Note that this also covers the case of binary computation on  $\mathbb{Z}_2 = \mathbb{F}_2$ . For sampling a uniformly random value  $r$  from  $\mathcal{R}$ , we write  $r \leftarrow_{\$} \mathcal{R}$ .

We consider protocols based on linear secret sharing. By  $\llbracket x \rrbracket$ , we denote a private value  $x \in \mathcal{R}$  which is secret shared between the parties according to the specific secret sharing scheme used in the respective protocol. Linearity of the scheme enables to compute  $\llbracket ax + y + b \rrbracket$  given arbitrary shares  $\llbracket x \rrbracket, \llbracket y \rrbracket$  and public values  $a, b \in \mathcal{R}$  without interaction. Some of the considered protocols use different linear secret sharing schemes, in which case we use notation  $\langle \cdot \rangle$  with the above properties as an *intermediate sharing semantic*. As for parties, we refer to individual shares in a circular way, e.g., for a sharing  $\llbracket x \rrbracket$  consisting of  $x^1, \dots, x^n$ ,  $x^{i+1}$  corresponds to the next share after  $x^i$  and  $x^1$  if  $i = n$ .

Finally, we denote the statistical security parameter by  $\sigma$  and the computational security parameter by  $\kappa$ . Usual parameter choices are  $\sigma = 40$  and  $\kappa = 128$  which are consistent with [12, 14, 21, 28] where applicable and stated.

**Pseudorandom Functions (PRFs) for Joint, Non-interactive Randomness Generation.** The protocols considered in this paper use PRFs to enable one or more parties, holding a random PRF key of length  $\kappa$ , to query a deterministic function (parametrized by the key) with the results being computationally indistinguishable from the results when querying a uniform random function. For details and a formal definition regarding PRFs, see [23]. To enable any subset of parties to sample common random values  $r \leftarrow_{\$} \mathcal{R}$  non-interactively, we assume that each such subset has established its own independently random and pre-shared PRF key as in [12, 14, 21, 28].

<sup>1</sup> Some of the considered protocols are designed for  $\mathbb{Z}_{2^\ell}$ , but also work for finite fields.

**Collision-Resistant Hash Functions.** The considered works [12, 14, 21, 28] all utilize collision resistant hash functions. This is a family of functions  $H_k : \{0, 1\}^* \rightarrow \{0, 1\}^{\ell(\kappa)}$  ( $\ell(\kappa)$  is the output size) where for a key  $k$  generated by a probabilistic generator in time polynomial in  $\kappa$ , any efficient adversary with access to  $k$  is able to find a collision, i.e.,  $x \neq x'$  s.t.  $H_k(x) = H_k(x')$  only with probability negligible in  $\kappa$ . For details and a formal definition, we refer to [23]. Throughout this work, we use  $H$  as a hash function without a key as used throughout the protocols in [12, 14, 21, 28]. This can be interpreted as the parties jointly and securely computing a key  $k$  to use throughout a protocol execution. Furthermore, we denote the output length  $\ell(\kappa)$  by  $|H|$ .

**Security Model.** The protocols from [12, 14, 21, 28] considered in this work aim to be secure against an active, nonadaptive adversary  $\mathcal{A}$  corrupting one party, corresponding to an honest majority setting as the protocols are for three or four parties. They are in the secure channels setting where each pair of parties has a secure communication channel between them. Furthermore, they consider security in the simulation-based real-world/ideal-world paradigm [9] to formalize that an interactive  $n$ -party protocol  $\Pi$  securely implements a functionality  $\mathcal{F}$ . For completeness, we provide details on this security definition in [8, App. A].

We consider variants of the security model where the adversary can prevent other parties from receiving output (security with abort), where either all or no parties receive outputs (fairness), or where all parties receive outputs irrespective of the adversary’s behavior (robustness) [13, 18]. For modularization, we also use the *hybrid model* [9]. Details again are provided in [8, App. A].

### 3 Our Attack and Fix: Trident [12] as an Example

We begin by providing a detailed explanation of our attack against the four-party protocol Trident [12] as its simple and elegant protocol description provides a good starting point to elaborate on our attack which we will later show to generalize to other protocols. We denote the parties by  $P_1, P_2, P_3$ , consistent with [12], but rename their party  $P_0$  to  $P_h$  here, as it acts as a helper and is of no relevance for our attack. The security model in [12] is either security with abort or even fairness, and our attack works the same for both settings.

Trident uses different secret sharing semantics with the ones provided in Table 2 being the relevant ones for our work. As an *intermediate* sharing semantic, the replicated sharing  $\langle \cdot \rangle$  is used. The main sharing semantic in Trident then is  $\llbracket \cdot \rrbracket$  where a sharing  $\llbracket v \rrbracket$  can also be viewed as each party holding  $m_v$  and a  $\langle \cdot \rangle$ -sharing of a random mask  $\lambda_v$  s.t.  $m_v = v + \lambda_v$ . Note that the sharings are linear, i.e., given  $\llbracket \cdot \rrbracket$  respectively  $\langle \cdot \rangle$ -sharings of  $x, y \in \mathcal{R}$  and public  $a, b \in \mathcal{R}$ ,  $\llbracket \cdot \rrbracket$  respectively  $\langle \cdot \rangle$ -sharings of  $ax + y$  can be computed by applying these operations on each separate component of the sharings. Furthermore, a sharing of  $x + b$  can be computed by adding  $b$  to  $m_x$  in  $\llbracket \cdot \rrbracket$ -sharings or adding  $b$  to  $x^1$  in  $\langle \cdot \rangle$ -sharings. Trident uses *function-dependent preprocessing* where all  $\langle \cdot \rangle$ -sharings of masks  $\lambda_v$

**Table 2.** Secret sharing semantics for sharing a value  $v \in \mathcal{R}$  in Trident [12]. Party  $P_h$  holds  $\lambda_v^1, \lambda_v^2, \lambda_v^3$  for a sharing  $\llbracket v \rrbracket$  which we abstract away here as it is of no relevance for our attack.

Sharing Type	$P_1$	$P_2$	$P_3$	Correlation
$\langle v \rangle$	$(v^2, v^3)$	$(v^3, v^1)$	$(v^1, v^2)$	$v = v^1 + v^2 + v^3$
$\llbracket v \rrbracket$	$(\mathbf{m}_v, \lambda_v^2, \lambda_v^3)$	$(\mathbf{m}_v, \lambda_v^3, \lambda_v^1)$	$(\mathbf{m}_v, \lambda_v^1, \lambda_v^2)$	$\mathbf{m}_v = v + \lambda_v^1 + \lambda_v^2 + \lambda_v^3$

across the circuit are computed before inputs are provided by the protocol—the online phase only computes the  $\mathbf{m}_v$  values.

The core of Trident is its multiplication sub-protocol, which also provides the target for our attack. Linear operations are non-interactive, following the linearity of the  $\llbracket \cdot \rrbracket$ -sharings. Finally, for input and output phases, we simply refer to [12] as they play no role for our attack. To multiply two sharings  $\llbracket x \rrbracket, \llbracket y \rrbracket$ , observe the following using the linearity of  $\langle \cdot \rangle$ -sharings:

$$\langle x \cdot y \rangle = \langle (\mathbf{m}_x - \lambda_x) \cdot (\mathbf{m}_y - \lambda_y) \rangle = \mathbf{m}_x \cdot \mathbf{m}_y - \mathbf{m}_x \cdot \langle \lambda_y \rangle - \mathbf{m}_y \cdot \langle \lambda_x \rangle + \langle \lambda_x \cdot \lambda_y \rangle$$

In the preprocessing, we obtain  $\langle \gamma_{xy} \rangle$  with  $\gamma_{xy} = \lambda_x \lambda_y$ . While this is part of the multiplication protocol in [12], we abstract it away into a functionality  $\mathcal{F}_{\text{MultPre}}$  (as also used in, e.g., [28]), because its exact instantiation plays no role in our attack. Trident uses helper party  $P_h$  for instantiating  $\mathcal{F}_{\text{MultPre}}$  which we hence can ignore here. Then,  $\langle x \cdot y \rangle = \mathbf{m}_x \cdot \mathbf{m}_y - \mathbf{m}_x \cdot \langle \lambda_y \rangle - \mathbf{m}_y \cdot \langle \lambda_x \rangle + \langle \gamma_{xy} \rangle$  can be computed non-interactively by the linearity of  $\langle \cdot \rangle$ -sharings. The goal now is to obtain a sharing  $\llbracket z \rrbracket$  for  $z = x \cdot y$ . The parties non-interactively sample random shares for a mask  $\langle \lambda_z \rangle$  (using pre-shared keys). They then aim to obtain  $\mathbf{m}'_z := \mathbf{m}_z - \mathbf{m}_x \mathbf{m}_y$  from where they can non-interactively add  $\mathbf{m}_x \mathbf{m}_y$  to finally obtain  $\mathbf{m}_z$  and, hence, the output sharing  $\llbracket z \rrbracket$ .

$$\langle \mathbf{m}'_z \rangle = \langle x \cdot y + \lambda_z - \mathbf{m}_x \mathbf{m}_y \rangle = -\mathbf{m}_x \cdot \langle \lambda_y \rangle - \mathbf{m}_y \cdot \langle \lambda_x \rangle + \langle \gamma_{xy} \rangle + \langle \lambda_z \rangle$$

Now, the resulting sharing  $\langle \mathbf{m}'_z \rangle$  is opened interactively to obtain  $\mathbf{m}'_z$  in the clear, allowing a corrupted party to cheat by sending an incorrect or no message. Trident aims to detect such a cheating attempt by exploiting that each share of  $\langle \mathbf{m}'_z \rangle$  is known by two parties. It lets one of these parties send the value and lets the other one send a hash of the value for verification, using a collision-resistant hash function  $H$ . The receiver can then hash the plain value and check for equality. In case of a mismatch, it aborts. Hence, one party provides  $P_i$  with  $\mathbf{m}'_z^i$  while the other provides  $H(\mathbf{m}'_z^i)$ . In case of an incorrect  $\mathbf{m}'_z^i$  sent, its hash value will differ from the received hash except for negligible probability, leading to any cheating attempt being detected.

The full multiplication sub-protocol is provided in Fig. 1. Note that we decouple the verification here, but for now, assume that it is done in parallel with the remaining online phase for each individual multiplication.

**Protocol**  $\Pi_{\text{mult}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \rightarrow \llbracket z \rrbracket$  of Trident [12]

*Input:*  $\llbracket \cdot \rrbracket$ -shares of  $x, y \in \mathcal{R}$ .

*Output:*  $\llbracket \cdot \rrbracket$ -shares of  $z = x \cdot y$ .

**Preprocessing:**

1. Invoke  $\mathcal{F}_{\text{MultPre}}$  on  $\langle \lambda_x \rangle$  and  $\langle \lambda_y \rangle$  to obtain  $\langle \gamma_{xy} \rangle$  with  $\gamma_{xy} = \lambda_x \cdot \lambda_y$ .
2. Non-interactively generate  $\langle \lambda_z \rangle$  by  $P_{i-1}, P_{i+1}$  sampling  $\lambda_z^i \leftarrow_{\$} \mathcal{R}$  for  $i \in \{1, 2, 3\}$ .

**Online:**

1. Each party  $P_i$  locally computes:
  - $m_z^{i-1} = -m_x \lambda_y^{i-1} - m_y \lambda_x^{i-1} + \gamma_{xy}^{i-1} + \lambda_z^{i-1}$ .
  - $m_z^{i+1} = -m_x \lambda_y^{i+1} - m_y \lambda_x^{i+1} + \gamma_{xy}^{i+1} + \lambda_z^{i+1}$ .
2. Each party  $P_i$  sends  $m_z^{i-1}$  to  $P_{i-1}$ .
3. Each party  $P_i$  locally computes  $m_z = m_x m_y + \sum_{j=1}^3 m_z^j$ .

**Verify:**

1. Each party  $P_i$  sends  $h_{i+1} = H(m_z^{i+1})$  to  $P_{i+1}$ .
2. Each party  $P_i$  proceeds if  $h_i = H(m_z^i)$  with  $m_z^i$  as previously received from  $P_{i+1}$  in step 2 of the online phase and aborts otherwise.

**Fig. 1.** Multiplication protocol of Trident [12] (simplified preprocessing).

### 3.1 Security of Unoptimized Trident

The security proof of Trident is deferred to the full version [11] of the conference version [12]. Important here is how the simulation of the online and verification phases of multiplications works. According to [11], the simulator emulates the honest parties throughout the protocol by setting their inputs to zero at the start of the protocol and then following the protocol specification. Then, the value  $m_z^i$  received by a corrupt  $P_i$  in the online phase of a multiplication is simulated by handing  $m_z^i$  (from the emulation of  $P_{i+1}$ ) to the adversary on behalf of  $P_{i+1}$ . Regarding verification, the simulator hands  $H(m_z^i)$  to the adversary on behalf of  $P_{i-1}$ . For messages sent by the corrupt party  $P_i$ , the simulator receives them from the adversary and follows the protocol for the emulated, honest parties.

First, note that the adversary is forced to provide correct messages for each multiplication to avoid the protocol aborting immediately. Should it cheat, the first multiplication where cheating occurs will lead to inconsistent hashes in the multiplication verification except for negligible probability.

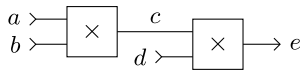
Now, note that the message  $m_z^i$  received by the adversary cannot be distinguished from a real-world execution of the same step. This is because the value is masked by  $\lambda_z^i$  freshly sampled using a PRF key unknown to the adversary—the existence of an efficient distinguisher would contradict the security of the PRF. The second message  $H(m_z^i)$  hence too cannot be distinguished from a real-world execution, as in both the real-world and the simulation, it simply is the hash of the other message. Yet, we observe that this only holds because both honest parties have consistent values  $m_z^i$ . This holds true if the verification is

done immediately as part of each multiplication, as any inconsistency between the honest parties must be caused by the adversary cheating due to the correctness of the protocol. Yet, in the case of cheating and immediate verification, any inconsistency in resulting shares would lead to an abort before the inconsistent shares can be used as input to other operations. We note that the argument regarding immediate verification is not provided or used in [11,12], an oversight that enables the issues discussed in the following. Yet, the entire proof in [11] appears to implicitly consider an immediate verification, not matching the optimized protocol description in [11,12], which is the topic of the next section.

### 3.2 Vulnerable Optimization: Delaying and Batching Verifications

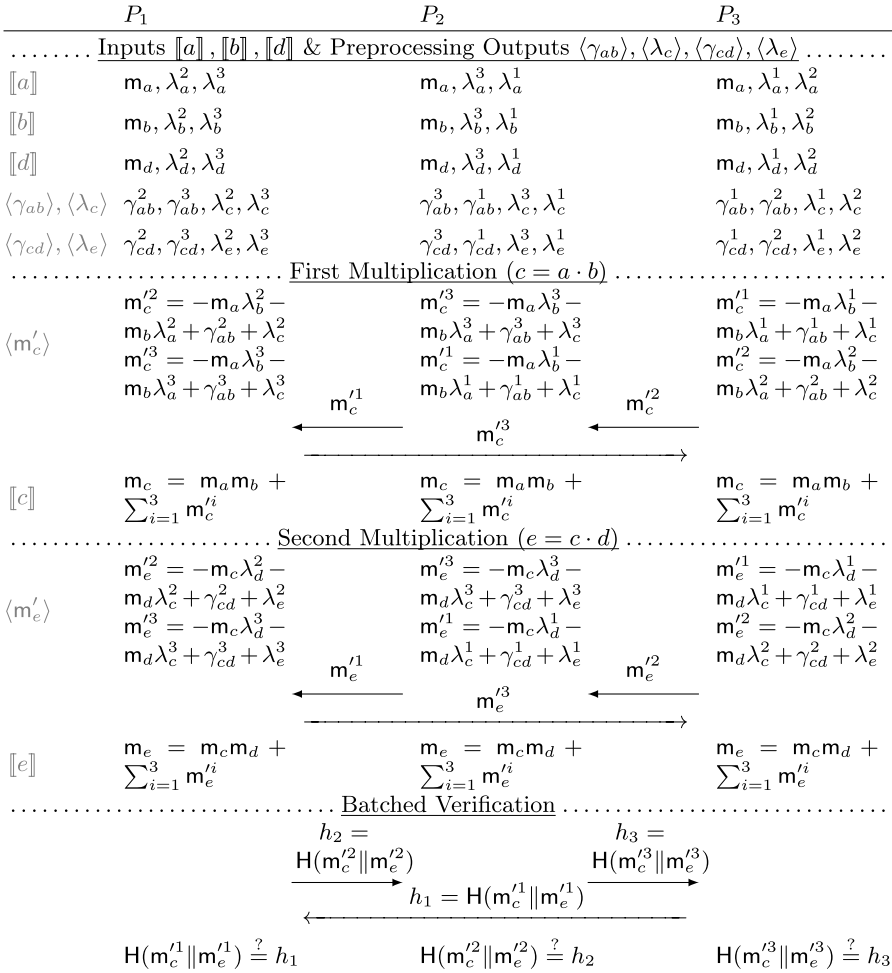
To achieve a negligible amortized communication overhead, the verification in Trident batches many individual verification steps from multiple multiplications with the communication of only a single verification. As an example, assume that three independent multiplications yield outputs  $\llbracket u \rrbracket, \llbracket v \rrbracket, \llbracket w \rrbracket$ . Then,  $P_i$  receives  $m_u^i, m_v^i, m_w^i$  from  $P_{i+1}$  which can be verified by  $P_{i-1}$  sending a single hash (instead of three separate ones)  $h_i = H(m_u^i || m_v^i || m_w^i)$  to  $P_i$  that then checks for consistency. This still maintains security due to the independence of the multiplications, preventing any inconsistency caused by cheating from influencing the input to other operations (cf. §3.1) besides verification.

Trident goes one step further by *optimistically* executing the protocol without any intermediate verification to then do a single batched verification immediately before any outputs are revealed to parties at the end of the protocol. This makes the verification communication overhead independent of the computed circuit and, hence, yields zero amortized overhead per multiplication. Yet, this optimization is considered only in the protocol specification, not in the security proof, and we will show in §3.3 how this renders the entire protocol insecure. As an example for such delayed verification, we compute  $c = a \cdot b$  and use the result to compute  $e = c \cdot d$  using the circuit in Fig. 2. Ignoring the input and output phases, the online phase of the protocol hence contains two subsequent online phases of  $\Pi_{\text{mult}}$  (Fig. 1) and the batched verification, jointly depicted in Fig. 3.



**Fig. 2.** Simple arithmetic circuit featuring two subsequent multiplications, providing a minimal example to enable our attack in §3.3.

It is easy to see that the batched verification still is sound—any initial incorrect message sent will be mirrored by the different, correct value as part of the input to  $H$ , causing at least one of the final comparisons to fail except for negligible probability. The issue arises on the side of privacy: Recall that if verification is not delayed, a party  $P_i$  will receive a value which can be simulated as shown



**Fig. 3.** Online phase of the two multiplications and batched verification required to evaluate the circuit from Fig. 2 in Trident [12], following the notation from Fig. 1.

before. Furthermore, it receives a hash which we simulate by taking the hash of the prior simulated value. If  $P_i$  is corrupt, both other parties are honest and hence provide consistent value and hash, yielding indistinguishability of our simulation. If  $P_i$  is not corrupt, one of the other parties may be, and inconsistent values may be received, but this does not matter as  $P_i$  is honest and hence, no simulation is needed.

Now, by delaying the verification, an incorrect message is not immediately detected. This unfortunately causes the honest parties to reconstruct inconsistent  $m_z$ , causing them to have inconsistent states. We proceed to show how a corrupt party can exploit that to cause the honest parties to provide inconsistent values

and hashes for a later multiplication in a way that allows the extraction of additional knowledge and, hence, breaking security.

### 3.3 Our Attack on Trident

Due to the symmetry of Trident, we assume  $P_1$  to be corrupt without loss of generality. We revisit the prior example of computing  $c = a \cdot b$  and  $e = c \cdot d$  (Fig. 2, Fig. 3). For the first multiplication,  $P_1$  is expected to send  $m_c^3$  to  $P_3$ . Our attack is to instead incorrectly send  $m_c^3 + 1$ . This will cause  $P_3$  to eventually recover  $m_c + 1$  instead of the correct value  $m_c$ . This error propagates to further values in the second multiplication. In particular,  $P_3$  has to compute  $m_e^1 = -m_c \lambda_d^1 - m_d \lambda_c^1 + \gamma_{cd}[1] + \lambda_e^1$ , but the “+1”-offset on  $m_c$  causes it to instead compute  $m_e^1 - \lambda_d^1$ . The error has not propagated to  $P_2$  yet which computes the correct  $m_e^1$  and sends it to  $P_1$ . In the verification,  $P_3$  receives  $h_3 = H(m_c^3 || \dots)$  from  $P_2$  which except for negligible probability is unequal to  $H(m_c^3 + 1 || \dots)$ . Hence, the cheating attempt is caught as soundness still holds. Yet, the corrupt party  $P_1$  receives from  $P_3$   $h_1 = H(m_c^1 || m_e^1 - \lambda_d^1)$ , not necessarily matching  $H(m_c^1 || m_e^1)$  given the values received from  $P_2$ .

**Formal Security Breach.** Assuming that the outputs the protocol provides to  $P_1$  and  $P_1$ ’s inputs do not allow to infer knowledge on  $d$ , e.g., by  $P_1$  receiving no output and  $d$  being another party’s input,  $h_1$  cannot be simulated. The simulator from [11] has a value  $\lambda_d^1$ , yet it uses zero-inputs for all honest parties, relying on the adversary not knowing the  $\lambda_v^1$  values that hence hide all shared values from the adversary (except for those known to the adversary by the design of the evaluated circuit). Assume w.l.o.g. that the simulator hands  $h_1 = H(m_c^1 || m_e^1 - \delta)$  to the adversary where  $\delta$  may depend only on what the simulator knows and, hence, not on value  $d$ . In a real execution with  $d = 0$ , the adversary receives  $h_1 = H(m_c^1 || m_e^1 - \lambda_d^1) = H(m_c^1 || m_e^1 - m_d + \lambda_d^2 + \lambda_d^3)$  and already has  $m_c^1, m_e^1, \lambda_d^2, \lambda_d^3$ . For  $d = 1$ , the adversary receives  $H(m_c^1 || m_e^1 - \lambda_d^1) = H(m_c^1 || m_e^1 + 1 - m_d + \lambda_d^2 + \lambda_d^3)$ . Hence, the distributions  $\{h_1, m_c^1, m_e^1, \lambda_d^2, \lambda_d^3\}$  are different for both cases  $d = 0$  and  $d = 1$  in a real execution except for the negligible probability of a hash collision. Yet, the corresponding distribution induced by the simulation contains  $h_1 = H(m_c^1 || m_e^1 - \delta)$  for  $\delta$  independent of  $d$ . Hence, it will be distinguishable from at least one of the distributions from the real execution, for  $d = 0$  or  $d = 1$ , and thus, the simulation is distinguishable from a real execution for some input(s).

**Concrete Attack.** Concretely, a corrupt  $P_1$  can now do an exhaustive search over all  $i \in \mathcal{R}$  to find an  $i$  s.t.  $H(m_c^1 || m_e^1 - i) = h_1$ . Except for the unlikely case of a hash collision, it then holds that  $i = \lambda_d^1$  and  $P_1$  that already has  $m_d, \lambda_d^2, \lambda_d^3$  can extract the plaintext  $d = m_d - \lambda_d^1 - \lambda_d^2 - \lambda_d^3$  that is another party’s input.

This brute-force attack only efficiently works for  $|\mathcal{R}| \ll 2^\kappa$ , which is given for usual domains, for sure in the binary domain, but even for, e.g.,  $\mathcal{R} = \mathbb{Z}_{2^{32}}$ . Also,  $\mathcal{R} = \mathbb{Z}_{2^{64}}$  is too small to properly mitigate such an attack, considering a usual choice of  $\kappa = 128$ . Furthermore, even for a very large ring  $\mathcal{R}$ ,  $P_1$  is

able to check if  $d = x$  by testing if  $H(\mathbf{m}_c^1 \| \mathbf{m}_e^1 + x - \mathbf{m}_d + \lambda_d^2 + \lambda_d^3) = h_1$ , i.e., if  $x - \mathbf{m}_d + \lambda_d^2 + \lambda_d^3 = \lambda_d^1 \Leftrightarrow d = x$ . Note that depending on the application and circuit,  $d$  might be from a much smaller subdomain of  $\mathcal{R}$ , perhaps even allowing for always efficient extraction of  $d$  on a large ring.

**Attacks for Larger Circuits.** While the circuit from Fig. 2 provides a minimal example for running our attack, attacks on significantly more complex circuits are also feasible. We observe that injecting an offset of 1 for any single targeted multiplication gate may lead to inconsistencies for all further multiplications that (transitively) depend on the output of the targeted gate, i.e., where there exists a path in the circuit from the targeted to the other multiplication gate. In all cases where only one multiplication depends on the targeted one, one of its input wires is the output of the targeted gate, and the other input wire is independent, we can attack the value of the other input wire. To see that, interpret Fig. 2 as part of a larger circuit where  $a, b, d$  may be not only inputs, but also outputs from other gates. We only require that intermediate wire  $c$  does not (transitively) connect to any other multiplication gate than that which outputs  $e$ , also implying that  $e$  can only be (transitively) used by linear gates. Then, the injected offset will not cause any inconsistency in the evaluation of multiplication gates beyond what we have used for our attack earlier. Hence, all other multiplications will contribute additional messages to the verification phase, but these will all be consistent. Thus,  $P_1$  receives  $h_1 = H(\dots \| \mathbf{m}_c^1 \| \dots \| \mathbf{m}_e^1 - \lambda_d^1 \| \dots)$  to compare to  $H(\dots \| \mathbf{m}_c^1 \| \dots \| \mathbf{m}_e^1 \| \dots)$  with the omitted parts being consistent. The attack works as for  $h_1 = H(\mathbf{m}_c^1 \| \mathbf{m}_e^1 - \lambda_d^1)$  to compare to  $H(\mathbf{m}_c^1 \| \mathbf{m}_e^1)$  before, simply with the consistent values injected in-between.

If the targeted multiplication has paths to multiple other multiplications, the attack becomes more complex, but persists. While the prior example already is sufficient to demonstrate Trident to be insecure, we also provide a further example for a more complex circuit topology in the full version [8, App. B].

### 3.4 First Attempts on Fixing Trident

One simple fix of Trident is to use the unoptimized multiplication where the verification is used for each individual multiplication. Moreover, as remarked in §3.2, the verification over multiple multiplications can even be batched as long as these multiplications are independent. For that, recall from §3.1 that security is maintained as long as the inputs to a multiplication are consistent between all parties. If we do one batched verification per layer of multiplications in a circuit, this still ensures that no inconsistent data can flow into another multiplication before being detected by a verification step. While fixing the issue that enabled our attack in §3.3, this requires at least  $d$  many verifications where each party sends a hash value for a circuit of multiplicative depth  $d$ . Hence, this solution results in an overhead that depends on the circuit, hence not amortizing to zero. This is efficient for circuits of very low depth. Yet, with increasing depth it becomes less efficient, adding more verification overhead. We revisit this issue in §3.5, demonstrating a fix that is efficient for circuits of any depth.

We also note that the part of the preprocessing that we abstracted away ( $\mathcal{F}_{\text{MultPre}}$  to multiply  $\langle \lambda_x \rangle$  and  $\langle \lambda_y \rangle$  to obtain  $\langle \gamma_{xy} \rangle$  with  $\gamma_{xy} = \lambda_x \cdot \lambda_y$ ) internally also uses the batched verification [12]. As all instances of  $\mathcal{F}_{\text{MultPre}}$  are used in parallel by [12], we note that running the batched verification for it at the end of preprocessing is sufficient and has zero amortized overhead, given the aforementioned considerations for the online phase. Hence, we continue to safely ignore the exact instantiation of  $\mathcal{F}_{\text{MultPre}}$  in our work.

Searching for a generally less expensive fix, recall that soundness of the batched and delayed verification in Trident still holds—the first incorrect message will always lead to a failed consistency check on the side of the, in this case, honest receiver. Hence, one may aim to do the consistency checks sequentially to cause this cheating attempt being caught and the protocol aborting before the corrupt party receives a hash. Yet, any of the parties may be corrupted, and by the symmetry of the protocol, any fixed, sequential order of consistency checks allows the adversary to corrupt the party receiving a hash first.

Hence, we might aim to completely hide the hash  $h_i$  from the party  $P_i$  that should receive it in Trident. Each consistency check could then still be implemented using a secure sub-protocol to decide if  $h_i$  matches the value expected by  $P_i$ , similar to [22]. Yet, this still does not suffice: Considering the prior attack for corrupt  $P_1$  on the simple circuit in Fig. 2, the outcome of the consistency check for  $h_1$  alone already reveals whether  $\lambda_d[1] = 0$  which would imply that  $d = m_d + \lambda_d^2 + \lambda_d^3$ , which is computable by corrupt  $P_1$ .

### 3.5 Fixing Trident

While Trident could be fixed with an overhead linear in the multiplicative depth of the circuit as discussed in §3.4, we now propose a solution with zero amortized overhead. Our core idea is simple: As proposed in §3.4, we do the consistency checks in any secure MPC protocol as sub-protocol of Trident. While we have seen that the output of an individual check still can leak information, we have also noted that overall soundness holds, i.e., at least one of the checks will always reject if a party cheats. In this case, we now aim to hide the outputs of the other consistency checks to avoid leakage. To this end, instead of running three individual consistency checks, we run a single one inside a secure sub-protocol that executes all three individual checks, but only reveals the AND over all outputs, i.e., if any of the checks failed, but not which specific one. Hence, the output will always be **Reject** if cheating occurred and always be **Accept** otherwise.

We introduce a functionality  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$  (formally specified in Fig. 4), which we keep generic to later also allow its utilization in further contexts to fix other protocols in the remainder of the paper. The functionality is parametrized by a set of equality checks  $A$  where for each check  $a \in A$ , parties  $P_{s_a}, P_{t_a}$  have a value each with the goal of checking for equality between both. In the current context of Trident (cf. Fig. 1), each party  $P_i$  originally computes a hash  $h_i^1$  over its received values while—in the original protocol—receiving hash  $h_i^2$  (originally labeled  $h_i$ ) to compare to from party  $P_{i-1}$ . We let both parties input  $h_i^1, h_i^2$  to the

functionality, requiring for Trident to choose  $A = \{1, 2, 3\}$  as labels for its three individual checks and  $s_i = i, t_i = i - 1$  for all  $i \in \{1, 2, 3\}$ , indicating that  $h_i^1, h_i^2$  are provided by  $P_i$  respectively  $P_{i-1}$ . For simplicity, we call the functionality with these specific parameters  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$ .

**Ideal Functionality**  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$

**Input:**  $h_a^1, h_a^2 \in \{0, 1\}^{|\mathbb{H}|}$  with  $h_a^1$  provided by  $P_{s_a}, h_a^2$  provided by  $P_{t_a}$ , for  $a \in A$ .  
**Output:** **Accept** if  $h_a^1 = h_a^2 \forall a \in A$ , **Reject** otherwise.

**Fig. 4.** Functionality for joined equality checking.

Using this, we fix Trident’s multiplication from Fig. 1 by replacing its original verification with that in Fig. 5, calling the resulting multiplication protocol  $\Pi_{\text{mult}}^{\text{FIXED}}$ . The “...” here stands for the respective  $\mathbf{m}_z^i, \mathbf{m}_z^{i+1}$  from all other multiplications as the batched check is done once over all multiplications in the end, before any output is revealed by the protocol.

**Protocol** Verification step of  $\Pi_{\text{mult}}^{\text{FIXED}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \rightarrow \llbracket z \rrbracket$  for Trident [12]

**Verify (batched):**

1. Each  $P_i$  sends  $h_i^1 = \text{H}(\dots \|\mathbf{m}_z^i\| \dots)$  and  $h_{i+1}^2 = \text{H}(\dots \|\mathbf{m}_z^{i+1}\| \dots)$  to  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$ .
2.  $P_i$  proceeds if it receives **Accept** from  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$  and aborts otherwise.

**Fig. 5.** Fixed, batched verification of the multiplication of Trident [12] in Fig. 1.

We first show that the resulting fixed version of Trident is secure using  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$  as a hybrid and then provide an instantiation for  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$ .

**Theorem 1.** *Trident [12] using the fixed multiplication protocol  $\Pi_{\text{mult}}^{\text{FIXED}}$  (cf. Fig. 5) is secure in the  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$ -hybrid model.*

*Proof.* We provide a proof sketch only as the security proof in [11] already covers most of the protocol, including all unchanged components. Proving security requires providing a simulator  $\mathcal{S}$ , given an adversary  $\mathcal{A}$  and a fixed corrupt party  $P_i$ . We do this in the following as a single proof over the whole protocol, as the delayed, batched verification allows inconsistent sharings (different  $\mathbf{m}_v$ -values per party) to propagate through the circuit, being detected only later after execution of the gate that they were introduced by. Throughout the proof,  $\mathcal{S}$  locally emulates the honest parties, keeping their internal state (except for their unknown inputs) and following the protocol specification.

First, the preprocessing is simulated following [11]. Also, it is easy to see that the protocol implements the desired functionality. Inputs are provided through

the protocols in [12], where  $\mathcal{S}$  lets honest parties share 0 instead of their actual inputs and can extract the corrupt party’s inputs as described in [11].

The simulator then follows the circuit’s topology, using the shares held by the individual parties and noting that different parties might also have different, inconsistent  $m_v$ -values introduced by cheating. Additions and multiplications by constants are handled locally on all shares. The remaining multiplications are simulated as follows:  $\mathcal{S}$  hands  $m_z^i$  to  $\mathcal{A}$  on behalf of  $P_{i+1}$ . Note that this message is masked by  $\lambda_z[z]$ , sampled using a random PRF key known to both honest parties, but not corrupt  $P_i$  and hence, not  $\mathcal{A}$ . Thus, distinguishing between the simulation and protocol execution from this message would also yield a distinguisher for the used PRF, violating the security assumption regarding the PRF.  $\mathcal{S}$  receives from  $\mathcal{A}$  a value  $m_z^{i-1}$  (not necessarily consistent with  $m_z^{i-1}$  computed internally on behalf of  $P_{i+1}$ ) or nothing in which case it sends abort to  $\mathcal{F}$ , simulates the other parties aborting and outputs what  $\mathcal{A}$  outputs.  $\mathcal{S}$  uses this  $m_z^{i-1}$  intended for  $P_{i-1}$ , emulating  $P_{i-1}$ ’s behavior using the received value to compute its  $m_z$ .

In the batched verification,  $\mathcal{S}$  receives from  $\mathcal{A}$  values  $h_i^1, h_{i+1}^2 \in \{0, 1\} \in \{0, 1\}^{|\mathcal{H}|}$  that  $P_i$  is supposed to send to  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$ . If it does not receive these values, it simulates an **abort** as above. Then,  $h_{i-1}^1, h_{i+1}^1, h_{i-1}^2, h_i^2$  are computed by  $\mathcal{S}$  on the shares of their respective honest parties. If  $h_j^1 \neq h_j^2$  for any  $j \in \{1, 2, 3\}$ , then  $\mathcal{S}$  hands **Reject** to  $\mathcal{A}$  on behalf of  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$ , sends **abort** to  $\mathcal{F}$ , simulates the other parties aborting and outputs what  $\mathcal{A}$  outputs. Otherwise, it sends **Accept** to  $\mathcal{A}$  on behalf of  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$  and proceeds.

To see why this properly simulates the protocol, note that if  $P_i$  cheats in any multiplication, it does so the first time for some multiplication with output  $\llbracket z \rrbracket$  by sending some  $m_z^{i-1} + \delta, \delta \in \mathcal{R}, \delta \neq 0$  instead of  $m_z^{i-1}$ . As all prior execution was done without cheating by definition, the inputs to the multiplication have consistent shares between all parties. Hence,  $P_{i+1}$  computes correct  $m_z^{i-1}$ . Now,  $P_{i+1}$  includes  $m_z^{i-1}$  in  $h_{i-1}^2$  while  $P_{i-1}$  includes the received  $m_z^{i-1} + \delta$  in  $h_{i-1}^1$ . Then, except for negligible probability of a hash collision,  $h_{i-1}^1 \neq h_{i-1}^2$  so that in the ideal and in the real world, the output of  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$  will be **Reject**.

Otherwise, there is no cheating during the multiplications, and all parties have consistent shares. Thus, if  $\mathcal{A}$  provides  $h_i^1, h_{i+1}^2$  honestly (it has all required values to compute these values), the output of  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$  will be **Accept** as the honest parties compute these hashes on the same input values, given the consistency of all shares. Otherwise, the output of  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$  will be **Reject** except for negligible probability, independent of which incorrect  $h_i^1$  or  $h_{i+1}^2$  is chosen exactly.

The output phase is simulated as in [11] and  $\mathcal{S}$  outputs what  $\mathcal{A}$  outputs.  $\square$

**Instantiating  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$ .** It remains to provide a secure instantiation for generic  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$ , yielding one for  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$  too. For that, we require secure evaluation of arithmetic circuits on the domain  $\mathcal{R} = \mathbb{F}_{2^{|\mathcal{H}|}}$ , which can be achieved by any secure MPC protocol in this setting. For now, we abstract this away to functionality  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$ , evaluating an arithmetic circuit  $\mathcal{C}$  on inputs provided

by  $n$  parties and, in addition,  $k$  uniform and secret random values:

$$\mathcal{F}_{\text{MPC}}^{\text{sub}}(\mathcal{C}, x_{1,1}, \dots, x_{1,j_1}, \dots, x_{n,1}, \dots, x_{n,j_n}) = \mathcal{C}(x_{1,1}, \dots, x_{1,j_1}, \dots, x_{n,1}, \dots, x_{n,j_n}, r_1 \leftarrow_{\$} \mathbb{F}_{2^{|\mathbf{H}|}}, \dots, r_k \leftarrow_{\$} \mathbb{F}_{2^{|\mathbf{H}|}}),$$

with arithmetic circuit  $\mathcal{C}$ , inputs  $x_{i,1}, \dots, x_{i,j_i} \in \mathbb{F}_{2^{|\mathbf{H}|}}$  being provided by  $P_i$  for  $1 \leq i \leq n$  ( $j_i$  is the number of inputs by  $P_i$ ), and arbitrary, but fixed  $k \in \mathbb{N}_0$ . We will later instantiate  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$  for the specific case of  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$ , utilizing that Trident is designed for evaluating arithmetic circuits too.

Recall that our goal is to check if  $h_a^1 = h_a^2$  for all  $a \in A$  at once. We aim to do that efficiently using a small circuit and  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$ . A simple random linear combination check fits this requirement, having a multiplicative depth of only 1. The resulting instantiation is provided in Fig. 6 and the exact idea of the random linear combination check will become clear in the subsequent proof. Note that it requires computing on a field, which is why we interpret all input hashes as elements of  $\mathbb{F}_{2^{|\mathbf{H}|}}$ . Note that this field must be and indeed is large, s.t. soundness is violated with probability negligible in the statistical security parameter  $\sigma$ . We then compute a random linear combination using  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$  as defined before, output the result  $x$  to all parties, and then **Accept** if  $x = 0$ .

**Protocol**  $\Pi_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$

**Input:**  $h_a^1, h_a^2 \in \{0, 1\}^{|\mathbf{H}|}$  (interpreted as elements of  $\mathbb{F}_{2^{|\mathbf{H}|}}$ ) with  $h_a^1$  provided by  $P_{s_a}$ ,  $h_a^2$  provided by  $P_{t_a}$ , for  $a \in A$ .

**Output:** **Accept** if  $h_i^1 = h_i^2 \forall a \in A$ , **Reject** otherwise.

1. Compute  $x = \sum_{a \in A} r_a \cdot (h_a^1 - h_a^2)$  for  $r_a \leftarrow_{\$} \mathbb{F}_{2^{|\mathbf{H}|}}, a \in A$ , using  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$ .
2. **Accept** if  $x = 0$ , **Reject** otherwise.

**Fig. 6.** Protocol instantiating  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$ .

**Theorem 2.** Protocol  $\Pi_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$ , given in Fig. 6, securely instantiates  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$  (Fig. 4) in the  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$ -hybrid model, assuming that  $|\mathbf{H}| > \sigma$ .

*Proof.* For value  $x$  computed by  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$ , it holds that  $x = \sum_{a \in A} r_a \cdot (h_a^1 - h_a^2)$ . If  $h_a^1 = h_a^2$  for all  $a \in A$ , it trivially follows that  $x = 0$  and the protocol outputs **Accept**. Otherwise, let  $\hat{a} \in A$  be such that  $h_{\hat{a}}^1 \neq h_{\hat{a}}^2$  which implies that  $h_{\hat{a}}^1 - h_{\hat{a}}^2 \neq 0$ . If still  $x = 0$ , then

$$r_{\hat{a}}(h_{\hat{a}}^1 - h_{\hat{a}}^2) = - \sum_{a \in A \setminus \{\hat{a}\}} r_a \cdot (h_a^1 - h_a^2) \implies r_{\hat{a}} = - \frac{\sum_{a \in A \setminus \{\hat{a}\}} r_a \cdot (h_a^1 - h_a^2)}{h_{\hat{a}}^1 - h_{\hat{a}}^2}.$$

Note that  $r_{\hat{a}}$  is sampled uniformly at random from  $\mathbb{F}_{2^{|\mathbf{H}|}}$ , independently of all other  $r_a$  and  $h$  values. Then, the probability of it satisfying the prior equation

and hence accepting is  $|\mathbb{F}_{2^{|H|}}|^{-1} = 2^{-|H|} \leq 2^{-\sigma}$  and, hence, negligible. Thus, the protocol correctly instantiates the desired functionality.

Regarding simulation,  $\mathcal{S}$  receives from  $\mathcal{A}$  all of its inputs to  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$  or, if no or insufficient input is provided, sends **abort** to  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$ , simulates the other parties aborting, and outputs what  $\mathcal{A}$  outputs. It sends  $\mathcal{A}$ 's inputs to  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$ , receiving back either **Accept** or **Reject**. If  $\mathcal{S}$  receives **Accept**, then it provides 0 to  $\mathcal{A}$  on behalf of  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$ . Otherwise, it samples uniformly random  $x' \in \mathbb{F}_{2^{|H|}}$  and provides that to  $\mathcal{A}$  on behalf of  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$ . Finally, it outputs what  $\mathcal{A}$  outputs.

Note that the only difference between ideal and real world is that in the case of a **Reject**, the real  $x$  might differ from  $x'$  sampled by  $\mathcal{S}$ . Yet there must exist an  $\hat{a}$  such that  $h_{\hat{a}}^1 \neq h_{\hat{a}}^2$  as  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$  outputs **Reject**, and

$$x = \sum_{a \in A} r_a \cdot (h_a^1 - h_a^2) = r_{\hat{a}}(h_{\hat{a}}^1 - h_{\hat{a}}^2) + \sum_{a \in A \setminus \{\hat{a}\}} r_a \cdot (h_a^1 - h_a^2).$$

For any fixed  $r_a$  for  $a \in A \setminus \{\hat{a}\}$ , the mapping  $r_{\hat{a}} \mapsto r_{\hat{a}}(h_{\hat{a}}^1 - h_{\hat{a}}^2) + \sum_{a \in A \setminus \{\hat{a}\}} r_a \cdot (h_a^1 - h_a^2)$  is bijective as  $h_{\hat{a}}^1 - h_{\hat{a}}^2 \neq 0$  is invertible in  $\mathbb{F}_{2^{|H|}}$ . As  $r_{\hat{a}}$  is uniformly random, so is  $x$ , proving that  $x$  and  $x'$  cannot be distinguished.  $\square$

**Instantiating  $\Pi_{\text{mult}}^{\text{FIXED}}$ .** Now, it easily follows that Trident, using  $\Pi_{\text{mult}}^{\text{FIXED}}$  (Fig. 5), is secure in the plain-model if we instantiate  $\mathcal{F}_{\text{CHECKEQS}}^{\text{Trident}}$  with  $\Pi_{\text{CHECKEQS}}^{\text{Trident}}$  (Fig. 6) and use a secure instantiation of  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$  in Trident's setting.

$\mathcal{F}_{\text{MPC}}^{\text{sub}}$  can be instantiated in this setting using any secure MPC protocol for the same setting that supports arithmetic circuits over  $\mathbb{F}_{2^{|H|}}$  and generating shares of random values to use inside a circuit. As remarked in §3.4, Trident is still secure if its original verification is not delayed, while this results in higher amortized communication cost. Furthermore, it is easy to generate a random sharing  $\llbracket r \rrbracket$  for  $r \leftarrow \mathcal{R}$  by for each  $i \in \{1, 2, 3\}$ ,  $P_{i-1}, P_{i+1}$  locally sampling  $\lambda_r[i] \leftarrow \mathcal{R}$  using pre-shared keys in the setup and setting  $m_r = 0$ . Finally, while Trident is designed for domain  $\mathcal{R} = \mathbb{Z}_{2^\ell}$  for performance reasons, it is easy to see that it works on any commutative ring  $\mathcal{R}$  and, hence, also  $\mathcal{R} = \mathbb{F}_{2^{|H|}}$ . Thus, we fix Trident with a new verification that is instantiated with the original, but unoptimized and thus secure Trident (i.e., with immediate verification) on a field.

Given that the verification step is independent of the computed circuit, we achieve the same zero amortized overhead for this step as (optimized) Trident did, but eliminate the security flaw of Trident. The generic  $\Pi_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$  has a constant multiplicative depth of only one, yielding minimal round overhead. More concretely, the immediate verification within  $\Pi_{\text{CHECKEQS}}^{\text{Trident}}$  is only required for three independent multiplications, hence it can be batched (§3.4). The verification only uses three multiplications, three subtractions, two additions, and generates three random shares. Thus, the concrete overhead is negligible and the use of  $\mathcal{R} = \mathbb{F}_{2^{|H|}}$  during verification comes at a negligible performance penalty.

### 4 Attacking and Fixing Fantastic Four [14]

We proceed to extend our attack on Trident [12] from §3 to the robust four-party protocol from Fantastic Four [14].<sup>2</sup> The attack is enabled by the protocol using the same idea for delayed, batched consistency checks as Trident [12]. Fantastic Four utilizes replicated secret sharing among parties  $P_1, P_2, P_3, P_4$  where a secret  $v \in \mathcal{R}$  is split into four random values  $v^1, v^2, v^3, v^4 \in \mathcal{R}$  subject to  $v^1 + v^2 + v^3 + v^4 = v$ . Each party  $P_i$  knows all shares  $v^j$  where  $j \neq i$ . We denote the sharing of  $v$  by  $\llbracket v \rrbracket$  and note that it is linear.

The intuition to multiply  $\llbracket x \rrbracket, \llbracket y \rrbracket$  is that

$$x \cdot y = \left( \sum_{i=1}^4 x^i \right) \cdot \left( \sum_{i=1}^4 y^i \right) = \sum_{i=1}^4 x^i y^i + \sum_{\{i,j\} \subseteq \{1,2,3,4\}, i \neq j} (x^i y^j + x^j y^i).$$

Each  $x^i y^i$  can be computed by three parties while for terms  $x^i y^j + x^j y^i$ , two parties can compute them and then communicate with the remaining parties. This communication can be verified, again using the fact that two parties know the value to be sent and at least one of them must be honest. The multiplication protocol of Fantastic Four is shown in Fig. 7.

Fantastic Four batches the verification at the end of optimistically executed segments which are not explicitly defined in [14]. Yet, in the implementation of the protocol in MP-SPDZ [24] (v0.4.2), the verification is also batched over multiple multiplications where the output of one influences others. Hence, there is an opportunity to deploy a similar attack to that on Trident discussed in §3.

We note that Fig. 7 does not specify for each subset  $\{g, h\} \subseteq \{1, 2, 3, 4\}$  who of the two parties is  $P_g$  and who is  $P_h$ . Similarly, it is not specified who is  $P_i$  and who is  $P_j$ , i.e., who sends  $z_{gh}^h$  to  $P_g$  and who later sends the hash in the verification phase. While [14] specifies that  $g < h$ , it still leaves open the roles of  $P_i$  and  $P_j$ . Hence, we use the role assignment specified in the implementation in MP-SPDZ [24]. We observe that the implementation does not strictly adhere to the  $g < h$  requirement of [14] (which does not affect the correctness and security of the protocol). For consistency, we will follow the role assignment of MP-SPDZ which we provide in Table 3.

We again use our minimal example for computing  $c = a \cdot b$  and  $e = c \cdot d$  (Fig. 2) to show how our attack from §3 translates to Fantastic Four. We assume here a corrupt  $P_3$ . The attack described in the following is also depicted in Fig. 8, highlighting how an injected error propagates through the protocol execution and who sends which messages. In the first multiplication  $c = a \cdot b$ , we let  $P_3$  send  $c_{12}^1 + 1$  instead of  $c_{12}^1 = a^2 b^1 + a^1 b^2 - c_{12}^2$ , where  $c_{12}^2 \leftarrow \mathcal{R}$ . This leads to  $P_2$  having output share  $c^1 + 1$  instead of  $c^1$ , given that the incorrect received message is one summand of the share. For multiplication  $e = c \cdot d$ ,  $P_2$  is expected to compute  $e_{13}^1, e_{14}^4, e_{34}^4$ , only the last being independent of incorrect  $c^1 + 1$ . For

<sup>2</sup> For simplicity, we call the protocol “Fantastic Four” here while noting that [14] also contains a three-party protocol that is not relevant to this work.

**Protocol**  $\Pi_{\text{mult}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \rightarrow \llbracket z \rrbracket$  of Fantastic Four [14]

*Input:*  $\llbracket \cdot \rrbracket$ -shares of  $x, y \in \mathcal{R}$ .

*Output:*  $\llbracket \cdot \rrbracket$ -shares of  $z = x \cdot y$ .

1. For every  $\{g, h\} \subseteq \{1, 2, 3, 4\}, g \neq h$ , use parties  $P_i, P_j$  with  $i, j \notin \{g, h\}$  to generate  $\llbracket z_{gh} \rrbracket$  with  $z_{gh} = x^g y^h + x^h y^g$ .
  - $P_i, P_j, P_h$  non-interactively sample  $z_{gh}^g \leftarrow \mathcal{R}$ .
  - $P_i, P_j$  compute  $z_{gh}^h = x^g y^h + x^h y^g - z_{gh}^g$ .
  - $P_i$  sends  $z_{gh}^h$  to  $P_g$ .
  - Set  $z_{gh}^i = z_{gh}^j = 0$ .
2. Each party  $P_i$  sets  $z^j = x^j y^j + \sum_{\{g,h\} \subseteq \{1,2,3,4\}, g \neq h} z_{gh}^j$  for  $1 \leq j \leq 4, j \neq i$ .

**Verify:**

1. For every  $\{g, h\} \subseteq \{1, 2, 3, 4\}, g \neq h$  and parties  $P_i, P_j$  with  $i, j \notin \{g, h\}$  as above where  $P_i$  sends  $z_{gh}^h$  to  $P_g$ :
  - $P_j$  sends  $h_{gh} = H(z_{gh}^h)$  to  $P_g$ .
  - Party  $P_g$  proceeds if  $h_{gh} = H(z_{gh}^h)$  (with  $z_{gh}^h$  as previously received from  $P_i$ ).
  - In case of inequality, output error message to other parties and start cheater identification as described in [14].

**Fig. 7.** Four-party multiplication protocol of Fantastic Four [14] (some operations inlined and separated verification).

**Table 3.** Assignment of the parties' roles in the MP-SPDZ [24] (v0.4.2) implementation of Fantastic Four [14].

$\{g, h\}$	$P_g$ (receives share)	$P_h$ (samples share)	$P_i$ (sends share)	$P_j$ (sends hash)
$\{1, 2\}$	$P_2$	$P_1$	$P_3$	$P_4$
$\{1, 3\}$	$P_3$	$P_1$	$P_2$	$P_4$
$\{1, 4\}$	$P_1$	$P_4$	$P_2$	$P_3$
$\{2, 3\}$	$P_3$	$P_2$	$P_4$	$P_1$
$\{2, 4\}$	$P_4$	$P_2$	$P_1$	$P_3$
$\{3, 4\}$	$P_4$	$P_3$	$P_1$	$P_2$

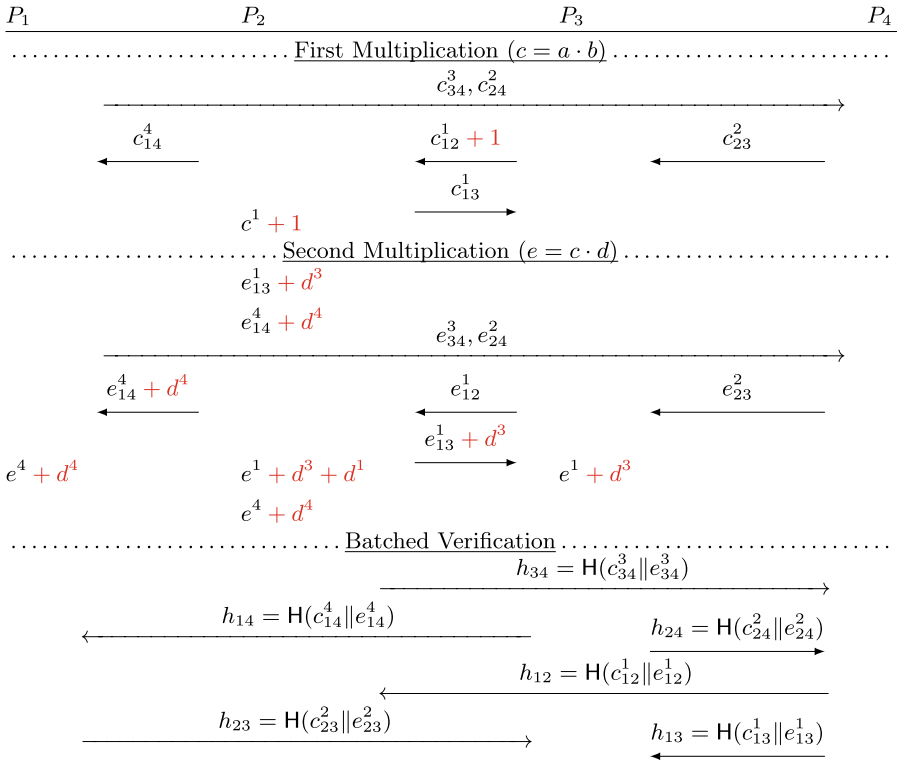
the other values, the offset on  $c^1$  causes it to compute

$$c^3 d^1 + (c^1 + 1)d^3 - e_{13}^3 = e_{13}^1 + d^3 \text{ instead of } e_{13}^1, \text{ and}$$

$$(c^1 + 1)d^4 + c^4 d^1 - e_{14}^1 = e_{14}^4 + d^4 \text{ instead of } e_{14}^4.$$

These are then sent to  $P_1$  respectively  $P_3$  and cause further errors in the shares for  $\llbracket e \rrbracket$  held by  $P_1, P_2, P_3$ .

It is easy to see that the consistency check will fail as  $P_2$  receives  $h_{12} = H(c_{12}^1 || e_{12}^1)$  from  $P_4$ , which except for negligible probability does not match  $H(c_{12}^1 + 1 || e_{12}^1)$  for the values received from  $P_3$ . Yet, we observe that the corrupt



**Fig. 8.** Two multiplications and batched verification required to evaluate the circuit from Fig. 2 in Fantastic Four [14]. Parts in red are errors introduced by cheating  $P_3$ , and we omit all locally held shares that do not contain any error. We also omit the comparison of hash values and cheater identification inside the batched verification, as they play no role in our attack.

party  $P_3$  receives  $c_{13}^1$  and  $e_{13}^1 + d^3$  from  $P_2$  while it receives  $h_{13} = H(c_{13}^1 || e_{13}^1)$  from  $P_4$ . Hence, similar to the attack in §3.3,  $P_3$  may for example check for different  $x \in \mathcal{R}$  if  $H(c_{13}^1 || e_{13}^1 + d^3 - x) = h_{13}$ . If it succeeds, then except for negligible probability it holds that  $x = d^3$  and  $P_3$  can extract  $d = d^1 + d^2 + d^3 + d^4$  as it already holds all other shares by the definition of the secret sharing scheme.

We also verified the aforementioned attack to check for a specific  $x \in \mathcal{R}$  by implementing a minimal proof-of-concept attack on the protocol implementation in MP-SPDZ [24]. Our code is available at <https://encrypto.de/code/F4attack>.

### 4.1 Fixing Fantastic Four

Fantastic Four can be fixed using the exact same approach as our fix for Trident in §3.5. The security flaw again stems from the fact that errors are allowed to propagate through the protocol execution, as outputs of multiplications can be used by further multiplications before being verified. Hence, we are able to

instead run an individual verification immediately after each layer of multiplications, decreasing the efficiency of the protocol.

To instead have the verification with zero amortized overhead, we first consider a simplified protocol providing fairness. This is easy to derive from Fantastic Four by simply replacing the cheater identification in the verification step with an immediate abort of the protocol, noting that this abort would always lead to no party receiving any output. Now, we instantiate the verification using  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$  (Fig. 4). Recall that for each  $\{g, h\} \subseteq \{1, 2, 3, 4\}, g \neq h$ , in the original verification (Fig. 7) one hash is sent and then compared. To capture all hash comparisons, we set  $A = \{12, 13, 14, 23, 24, 34\}$  and for each  $a \in A$ , let  $s_a$  be the index of the party sending  $h_a$  and  $t_a$  be the index of the party that receives it in Fig. 7. We call the functionality with these specific parameters  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$  for simplicity. The fixed verification step is provided in Fig. 9. It again renders the protocol secure, as except for negligible probability,  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$  exactly outputs **Accept** if no cheating occurred and **Reject** otherwise, following the same arguments as in §3.5. The proof is very similar to the one in §3.5, so we omit the details here.

**Protocol** Verification step of  $I_{\text{mult}}^{\text{FIXED}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \rightarrow \llbracket z \rrbracket$  for Fantastic Four [14]

**Verify (batched):**

1. For every  $\{g, h\} \subseteq \{1, 2, 3, 4\}, g \neq h$  and parties  $P_i, P_j$  with  $i, j \notin \{g, h\}$  as above where  $P_i$  sends  $z_{gh}^h$  to  $P_j$ :
  - $P_j$  sends  $h_{gh}^1 = \text{H}(\dots \|z_{gh}^h\| \dots)$  to  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$ .
  - $P_g$  sends  $h_{gh}^2 = \text{H}(\dots \|z_{gh}^h\| \dots)$  ( $z_{gh}^h$  as received from  $P_i$ ) to  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$ .
2.  $P_i$  proceeds if it receives **Accept** from  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$  and aborts otherwise.

**Fig. 9.** Fixed, batched verification of the multiplication of Fantastic Four [14] in Fig. 7.

To instantiate  $\mathcal{F}_{\text{MPC}}^{\text{sub}}$  required for the instantiation of  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$  (cf. §3.5), we can use Fantastic Four, but with immediate verification after each (layer of) multiplications, resulting in zero amortized overhead as it is used on a small circuit, independent of the circuit to be evaluated by the overall outer protocol.

### 4.2 Fixing the Robust Version of Fantastic Four

To provide robustness, Fantastic Four [14] is designed to identify honest parties in the case of a verification detecting any inconsistency. For classic robustness, they observe that  $P_g$  complaining about receiving a hash from  $P_j$  that is inconsistent with the value(s) received from  $P_i$  (cf. Fig. 7) indicates that one of these parties is cheating, immediately proving the uninvolved, fourth party to be honest. The protocol then lets all parties hand their shares to the honest party which locally finishes the computation. They also propose a “private robustness” variant which seeks to avoid disclosing clear text values to any party, even if proven honest.

There, a failed verification identifies a set of at most two parties containing the cheater. The protocol then removes parties in this set from the execution until it, in the worst case, ends up with the two parties running a semi-honest protocol, given that the malicious party must be among the previously identified parties.

While both approaches work when verification is not delayed, delaying the verification as in [14] not only breaks security, but also prevents reliable identification of one or multiple honest parties. As we have seen, inconsistencies introduced by a cheater that are not immediately detected cause honest parties to deviate from what an honest protocol execution would appear like. This can lead to honest parties sending incorrect values, which is detected during verification. As an example, note that in Fig. 8, honest  $P_2$  sends incorrect  $e_{13}^1 + d^3$  to  $P_3$  while honest  $P_4$  provides an inconsistent hash  $h_{13}$  that contains the correct  $e_{13}^1$  without any offset. In the full version of this paper [8, App. C], we provide more detailed examples where, e.g., both senders (of inconsistent value and hash) and the receiver are honest and demonstrate how both notions of robustness in Fantastic Four are broken by delaying verification.

Unfortunately, there appears to be no immediate fix with zero amortized overhead like for fairness (cf. §4.1). More precisely, note that while using  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$  detects a mismatch resulting from the first incorrect message sent, further mismatches can be produced by continuing to compute on inconsistent shares, and honest parties may send incorrect values. In  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$ , it would then be necessary to additionally identify the original, first inconsistency, which must be caused by the cheater. As  $\mathcal{F}_{\text{CHECKEQS}}^{\text{FantasticFour}}$  only receives hashes over data from all multiplications as input, it appears to be unfeasible to identify the exact multiplication causing an inconsistency. Hence, it seems not to be possible to reliably narrow down the cheater’s identity using the amortized approach.

A possible solution is to optimistically use the fixed, fair protocol and, should any cheating be detected at the end, run the layer-wise original verification round by round until the first inconsistency is detected. This ensures that the detected inconsistency is directly caused by the cheater, enabling the original robustness mechanisms. Note that this approach yields zero amortized overhead if no cheating occurs and a cheater is not able to abort the protocol execution. Yet, cheating would increase protocol execution cost, given that the round complexity is doubled by sequentially running all verification steps in the end. Given that robustness goes beyond our original attack and fix in §3, we leave it up to future work to formalize how robustness can be achieved securely and investigate if more efficient approaches exist.

## 5 SWIFT [28]: Almost-Evasion of Our Attack

The three-party protocol of SWIFT [28]<sup>3</sup> is another protocol using delayed verification checks and targeting robustness. The core idea (regarding multiplication)

<sup>3</sup> For simplicity, we call the protocol “SWIFT” here, noting that [28] also contains a four-party protocol that we do not examine in this paper.

of SWIFT is very close to that of Trident [12] (§3), except for evading the requirement for a fourth party to instantiate preprocessing multiplication  $\mathcal{F}_{\text{MultPre}}$ , and achieving robustness using a similar approach to cheater identification as that in §4.2 for Fantastic Four [14]. Yet, we note that the close similarity to Trident is present only in more recent rephrased and simplified versions of SWIFT, especially that in [7]. We decide here to target the original protocol in [28], noting that more recent versions are not consistent with each other and do not claim to resolve any security issue. For consistency with [28], note that we label the three parties  $P_0, P_1, P_2$  here, instead of starting with  $P_1$  as before.

The two secret sharing semantics important here are as defined in Table 4. The main sharing semantic is  $\llbracket \cdot \rrbracket$  and an intermediate additive sharing  $\langle \cdot \rangle$  between  $P_1, P_2$  is used. Both sharings are linear. The protocol uses *function-dependent preprocessing* where only the  $\beta_v$ -values are computed online.

**Table 4.** Secret sharing semantics for sharing a value  $v \in \mathcal{R}$  in SWIFT [28].

Sharing Type	$P_0$	$P_1$	$P_2$	Correlation
$\langle v \rangle$	—	$v^1$	$v^2$	$v = v^1 + v^2$
$\llbracket v \rrbracket$	$(\alpha_v^1, \alpha_v^2, \beta_v + \gamma_v)$	$(\alpha_v^1, \beta_v, \gamma_v)$	$(\alpha_v^2, \beta_v, \gamma_v)$	$\beta_v = v + \alpha_v^1 + \alpha_v^2$

The multiplication protocol of SWIFT is depicted in Fig. 10. It already includes optimizations that [28] only describes in text. Importantly, [28] notes that all verification is delayed and batched and that  $P_1, P_2$  can optimistically evaluate the circuit (up to the output phase) alone. Only in the verification,  $P_0$  receives all  $\beta_z + \gamma_z$  computed throughout the circuit and immediately verifies their consistency. It then locally computes  $\beta_z^{*1}, \beta_z^{*2}$  used by  $P_1, P_2$  before to send hashes, allowing  $P_1, P_2$  to verify consistency of the messages they have previously exchanged in the online phase. Note that  $P_0$  must receive the  $\beta_z + \gamma_z$  values first, as it has no knowledge of these if coming out of a multiplication otherwise, while requiring them for computing the hashes for later multiplications, depending on the outcome of the prior one. Finally, the hashes are again computed over the messages over all multiplications to leverage amortization.

We proceed by first discussing in §5.1 how SWIFT almost evades our attack from §3.3, still leaving a gap in its security proof, by making communication more asymmetric, using that  $P_0$  is inactive in the online phase before verification. Then, we show in §5.2 how a modified attack can still be applied, depending on how an underspecified part of the protocol is interpreted.

### 5.1 Unsuccessful Attack and Gap in the Security Proof

As an example, the flow of messages for computing  $c = a \cdot b$  and  $e = c \cdot d$  (Fig. 2) is given in Fig. 11. In case of corrupt  $P_0$ , our attack from §3.3 cannot be adapted to SWIFT because this party only sends hashes, which will exactly cause a mismatch for one of the other parties if one of these hashes is incorrect by the direct

**Protocol**  $\Pi_{\text{mult}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \rightarrow \llbracket z \rrbracket$  of SWIFT [28]

*Input:*  $\llbracket \cdot \rrbracket$ -shares of  $x, y \in \mathcal{R}$ .

*Output:*  $\llbracket \cdot \rrbracket$ -shares of  $z = x \cdot y$ .

**Preprocessing:**

1. Non-interactively generate  $\langle a_z \rangle$  by  $P_0, P_i$  sampling  $a_z^i \leftarrow \mathcal{R}$  for  $i \in \{1, 2\}$ .
2.  $P_1, P_2$  non-interactively sample  $\gamma_z \leftarrow \mathcal{R}$ .
3. Compute further preprocessing material (check [28] for details):
  - $\langle \chi \rangle$  where in addition,  $P_0$  has both shares  $\chi^1, \chi^2$ .
  - $P_1, P_2$  have  $\psi \in \mathcal{R}$ .
  - Prior values are random, subject to  $\chi^1 + \chi^2 + \psi = \gamma_x \alpha_y + \alpha_x \gamma_y + \alpha_x \alpha_y$ .

**Online:**

1. Party  $P_i$  for  $i \in \{1, 2\}$  computes  $\beta_z^{*i} = -(\beta_x + \gamma_x)\alpha_y^i - (\beta_y + \gamma_y)\alpha_x^i + \alpha_z^i + \chi^i$ .
2.  $P_1, P_2$  exchange  $\beta_z^{*1}, \beta_z^{*2}$ .
3.  $P_1, P_2$  compute  $\beta_z = \beta_z^{*1} + \beta_z^{*2} + \beta_x \beta_y + \psi$ .

**Verify:**

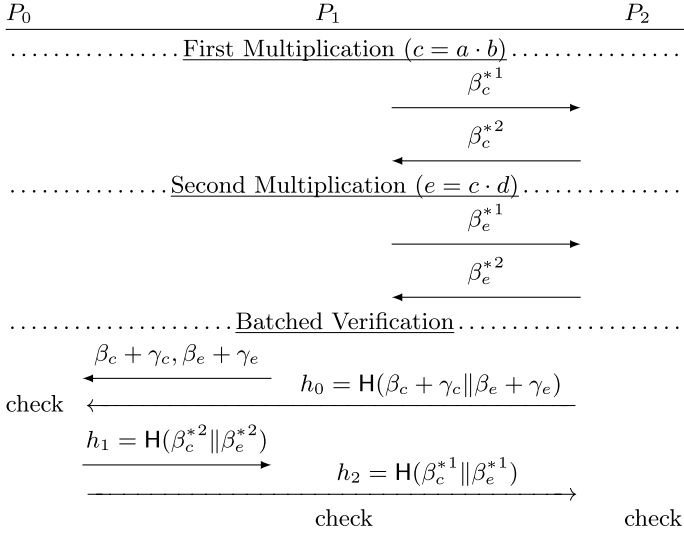
1.  $P_1$  sends  $\beta_z + \gamma_z$  and  $P_2$  sends  $h_0 = \mathbf{H}(\beta_z + \gamma_z)$  to  $P_0$ .
2. If  $P_0$  detects that  $h_0 \neq \mathbf{H}(\beta_z + \gamma_z)$  with  $\beta_z + \gamma_z$  as received by  $P_1$ , it outputs an error message and starts the cheater identification described in [28]. The steps below are not executed in that case.
3.  $P_0$  computes  $\beta_z^{*1}, \beta_z^{*2}$  as  $P_1, P_2$  did in step 1 of the online phase.
4.  $P_0$  sends  $h_i = \mathbf{H}(\beta_z^{*3-i})$  to  $P_i$  for  $i \in \{1, 2\}$ .
5. If  $P_i$  for  $i \in \{1, 2\}$  detects that  $h_i \neq \mathbf{H}(\beta_z^{*3-i})$  with  $\beta_z^{*3-i}$  as received by  $P_{3-i}$ , it outputs an error message and starts the cheater identification described in [28].

**Fig. 10.** Multiplication protocol of SWIFT [28] (some operations inlined and restructured according to the optimizations that [28] only describes in text).

action of  $P_0$ . Yet, it appears that the more sophisticated structure of SWIFT, allowing  $P_0$  to remain absent during most of the protocol execution, also thwarts our attack for another corrupted party, as we will describe in the following.

Assume a corrupt  $P_1$ . Furthermore, for a multiplication  $x \cdot y = z$ , assume that  $P_1, P_2$  hold consistent values  $\hat{\beta}_x, \hat{\beta}_y$  on their input wires which do not have to necessarily match the  $\beta_x, \beta_y$  that they would hold in an honest protocol execution. Now, in a multiplication,  $P_1$  uses  $\hat{\beta}_z^{*1} + \delta = -(\hat{\beta}_x + \gamma_x)\alpha_y^1 - (\hat{\beta}_y + \gamma_y)\alpha_x^1 + \alpha_z^1 + \delta$  for  $\delta \in \mathcal{R}$  that  $P_1$  can use to introduce an error to the message.  $P_2$  honestly sends  $\hat{\beta}_z^{*2} = -(\hat{\beta}_x + \gamma_x)\alpha_y^2 - (\hat{\beta}_y + \gamma_y)\alpha_x^2 + \alpha_z^2 + \chi^2$ . This will cause both parties to eventually compute  $\hat{\beta}_z = (\hat{\beta}_z^{*1} + \delta) + \hat{\beta}_z^{*2} + \hat{\beta}_x \hat{\beta}_y + \psi$ , offsetting the result by  $\delta$ . Still, it yields a perhaps incorrect, but consistent  $\hat{\beta}_z$  to both parties. For the example circuit from Fig. 2,  $P_1$  could use an error  $\delta \neq 0$  in the first multiplication. The parties  $P_1, P_2$  would then obtain  $\hat{\beta}_c = \beta_c + \delta$ .<sup>4</sup> In

<sup>4</sup> Of course, from this,  $P_1$  can also derive  $\beta_c$  given that it selects the error  $\delta$ .



**Fig. 11.** Online phase of the two multiplications and batched verification required to evaluate the circuit from Fig. 2 in SWIFT [28].

the next multiplication, corrupt  $P_1$  would receive

$$\begin{aligned} \hat{\beta}_e^{*2} &= -(\hat{\beta}_c + \gamma_c)\alpha_d^2 - (\hat{\beta}_d + \gamma_d)\alpha_c^2 + \alpha_e^2 + \chi^2 \\ &= -(\beta_c + \delta + \gamma_c)\alpha_d^2 - (\beta_d + \gamma_d)\alpha_c^2 + \alpha_e^2 + \chi^2 = \beta_e^{*2} - \delta\alpha_d^2 \end{aligned}$$

from  $P_2$ . Note that this message is still masked by  $\alpha_e^2$ .

The difficulty is that the verification towards  $P_0$  is done first. Recall that  $P_1$  can cause  $P_2$  to reconstruct incorrect  $\hat{\beta}_z$  along the computation, but it will also learn the same incorrect values. As  $P_2$  sends the hash over all  $\hat{\beta}_z + \gamma_z$  to  $P_0$ ,  $P_1$  can either send consistent, incorrect  $\hat{\beta}_z + \gamma_z$ , or send other values which except for negligible probability will lead to  $P_0$  detecting an inconsistency and starting cheater identification instead of proceeding with the verification step. This does not allow  $P_1$  to learn anything, as it knows the values that  $P_2$  computes the hash over and, hence, cannot use  $P_0$ 's reaction as an oracle to derive new information.

In §3.3, our attack relied on  $P_1$  receiving from one party a message,  $\hat{\beta}_e^{*2} = \beta_e^{*2} - \delta\alpha_d^2$  in this case, and from the other party a hash, using an inconsistent message as input, in this case  $\beta_e^{*2}$ . Yet,  $P_1$  is forced for each multiplication  $x \cdot y = z$  to send the same incorrect  $\hat{\beta}_z + \gamma_z$  that  $P_2$  has to  $P_0$  so that  $P_0$  does not detect an inconsistency. Hence,  $P_0$  uses the same inconsistency used by  $P_2$ . Specifically,  $P_0$  will have inputs  $\hat{\beta}_x + \gamma_x, \hat{\beta}_y + \gamma_y$ , including errors from prior multiplications influencing the input wires but matching the shares of  $P_2$ . Then,  $P_0$  will compute

$$\hat{\beta}_z^{*2} = -(\hat{\beta}_x + \gamma_x)\alpha_y^2 - (\hat{\beta}_y + \gamma_y)\alpha_x^2 + \alpha_z^2 + \chi^2$$

and include this in hash  $h_1$  to send to  $P_1$ , but this will be consistent with  $\hat{\beta}_z^*$  that  $P_1$  receives from  $P_2$ , not disclosing any additional information. For corrupt  $P_2$ , the aforementioned arguments are symmetrical.

We note that the prior arguments are neither used in [28] nor in the full version [27] that it refers to for the security proof. Instead, like for Trident [12] as discussed in §3.1, it appears that [27] proves security only for the unoptimized, incomplete protocol without any delayed and batched verification. Hence, the security proof is incomplete. We stress that our prior observations give reason to believe that the multiplication protocol of SWIFT [28] still is secure, but our prior arguments represent only a possible starting point for trying to provide a complete and correct security proof. They do not immediately yield a proof, and while our prior attack did not work, this does not prove SWIFT to be secure. We leave a fix to the security proof to future work. Should it not be possible to fix SWIFT in this way, another option would be the use of  $\mathcal{F}_{\text{CHECKEQS}}^{A_i, (s_a)_{a \in A}, (t_a)_{a \in A}}$  as in §3.5, yielding at least a fair protocol, noting prior difficulties in §4.2 to also reach robustness with this approach.

## 5.2 Potential Attack on SWIFT, Involving the Input Phase

As seen in §5.1, corrupt  $P_1$  and  $P_2$  appear to be unable to exploit causing both honest parties to reach inconsistent states because the honest  $P_0$  will detect any inconsistency before being able to send an inconsistent hash back to the corrupt party. Hence, we now try again to target  $P_0$ . For multiplications it only interacts with  $P_1, P_2$  during the verification, sending hashes to both parties and being unable to successfully attack the protocol.

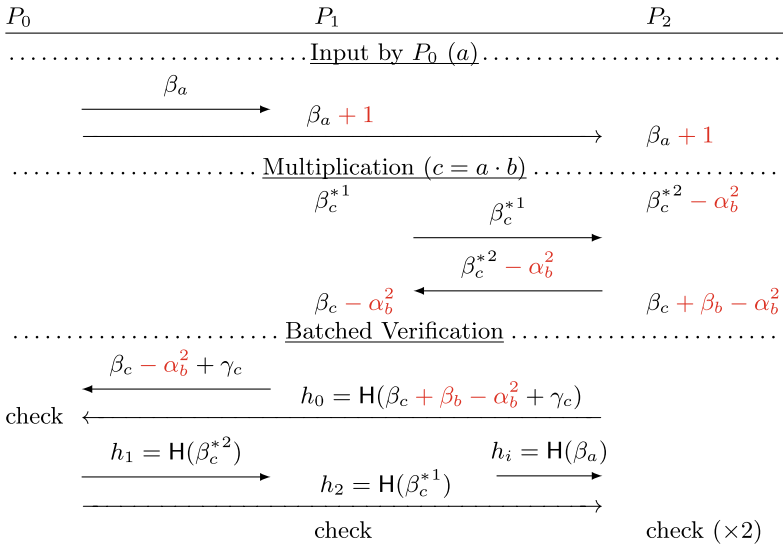
Instead, we investigate the input phase of SWIFT. As described in [28],  $P_0$  can share an input  $v \in \mathcal{R}$  as follows:

1. Non-interactively generate  $\langle a_v \rangle$  by  $P_0, P_i$  sampling  $a_v^i \leftarrow \mathcal{R}$  for  $i \in \{1, 2\}$  during preprocessing.
2. All parties non-interactively sample  $\gamma_v \leftarrow \mathcal{R}$  during preprocessing.
3.  $P_0$  computes  $\beta_v = v + a_v^1 + a_v^2$  (and uses  $\beta_v + \gamma_v$  as part of its share).
4.  $P_0$  sends  $\beta_v$  to  $P_1$ .
5.  $P_0, P_1$  “jmp-send  $\beta_v$  to  $P_2$ ” [28, p. 2656].

The “jmp-send” operation denotes that one of the parties  $P_0, P_1$  sends  $\beta_v$  and the other sends  $H(\beta_v)$  to  $P_2$  that then can check for consistency in [28]. While the distribution of roles is ambiguous in the paper, the notation [28, Notation 3.1 and Fig. 1] suggests that the first mentioned party, i.e.,  $P_0$  sends the actual value  $\beta_v$ . It is the same primitive that [28] also uses for multiplications—in Fig. 10, we simply inlined it in the overall protocol description. Regarding delaying the consistency check, it is stated that “[t]he communication of hash is done once and for all from  $P_j$  to  $P_k$ ” [28, §3.1] and “while the *verify* for a fixed ordered pair of senders will be executed once and for all in the end” [28, §3.1] in the context of “jmp-send”. It is unclear if “in the end” refers to the entire protocol (before the output phase), like for delayed verifications of the multiplications, or only the final “jmp-send”-instance with specific two senders in fixed roles. Unfortunately, [28] provides

no public implementation to cross-check. For our attack, we assume that the verification is delayed until immediately before the output phase, just like for multiplications, which appears to be at least within the under-specified protocol description. Furthermore, we assume that the verification towards  $P_0$  remains to be executed first. This would render the input phase part of the optimistic protocol execution.

We now consider a minimal circuit that computes  $c = a \cdot b$  where corrupt  $P_0$  provides input  $a$ ,  $P_1$  provides  $b$ , and  $P_2$  receives  $c$ . Clearly,  $P_0$  should not gain any information about  $b$ . Then, we let  $P_0$  send  $\beta_a$  to  $P_1$  but  $\beta_a + 1$  to  $P_2$ , yielding the situation depicted in Fig. 12. We label the hash to check the consistency of input  $a$  by  $h_i$  and note that using that, the inconsistency would be detected. Yet,  $P_0$  receives  $\beta_c - \alpha_b^2 + \gamma_c$  and  $h_0 = H(\beta_c + \beta_b - \alpha_b^2 + \gamma_c)$ . As in §3.3, this enables  $P_0$  to test for certain or even fully extract  $\beta_b$  while it already has  $\alpha_b^1, \alpha_b^2$  by definition of the secret sharing scheme. Then, it can derive  $b = \beta_b - \alpha_b^1 - \alpha_b^2$ , the private input of  $P_1$ .



**Fig. 12.** One input (other correct and omitted), one multiplication and batched verification required to evaluate  $c = a \cdot b$  in SWIFT [28]. Parts in red are errors introduced by cheating  $P_0$  and we omit all locally held shares that do not contain any error.

We note that this attack can easily be made impossible by clearly requiring that consistency checks for the input phase run immediately after the input phase, still resulting in zero amortized overhead.

**Note on Related Protocols Tetrad [29] and SOCIUM [7].** The four-party protocol Tetrad [29] improves upon Trident [12] with the online phase of multiplications deviating from Tetrad mostly by using the same trick as SWIFT [28]

to let two parties run most of the online phase and a third party only joining for the final verification. In contrast to SWIFT, Tetrad has a separate consistency check for the input phase, and this check is not delayed. Hence, we presume that Tetrad evades the original attack from §3.3 similar to SWIFT, still leaving a gap in the security proof (cf. §5.1), and also that it is not vulnerable to the attack from §5.2 involving the input phase. The gap in the proof may result in some details not appearing to be crucial for security. Hence, small optimizations in implementations may appear to still be secure, but create vulnerabilities, see §7.

The three-party protocol SOCIUM from [7] is a modification of SWIFT where only one fixed, known party can be corrupted maliciously, while others cannot deviate from the protocol even when corrupted. It uses only one batched consistency check for messages from the known potentially cheating party. This check fails if any cheating occurred, and as there is no further consistency check, especially none with the potentially cheating party receiving redundant data, it seems that our attack does not apply to SOCIUM.

## 6 Attacking and Fixing Quad [21]

The last instance of our attack is on the fair four-party protocol Quad [21]. We note that the underlying ideas of Quad are quite similar to those in Trident [12] (discussed in §3), also using a sharing semantic  $\llbracket v \rrbracket$  consisting of a masked value  $m_v = v + \lambda_v$  and a mask  $\lambda_v$  shared between the parties using a secondary sharing  $\langle \lambda_v \rangle$ . Multiplication accordingly follows the same intuition as in Trident, and the protocol is optimistically executed while hash-based consistency checks are executed once before revealing any outputs.

This opens exactly the same vulnerability described in §3, also allowing us to deploy a similar fix. For the concrete attack, it again remains to survey how an error introduced by a cheater propagates through the following computation in the exact protocol, enabling the cheater to receive inconsistent data from which it can extract information. This is essentially a technicality that reuses ideas of our attack and fix from §3, used throughout this paper. We provide full details on how the attack and fix exactly translate to Quad in the full version [8, App. D].

## 7 Responsible Disclosure

We found the vulnerable protocols to be implemented in the MPC frameworks MP-SPDZ<sup>5</sup> [24], HPMPC<sup>6</sup> [20, 21], and in the oblivious analytics system ORQ<sup>7</sup> [4]. The frameworks inherited the vulnerabilities from the insecure protocols at the time of disclosure. After acceptance of our paper, we informed the authors of all affected protocols and implementations on January 30, 2026. We

<sup>5</sup> <https://github.com/data61/MP-SPDZ/>, v0.4.2: Fantastic Four [14].

<sup>6</sup> <https://github.com/chart21/hpmpc/>, a9809dc: Fantastic Four [14] and Quad [21].

<sup>7</sup> <https://github.com/CASP-Systems-BU/orq/>, v1.0.0: Fantastic Four [14].

coordinated with the maintainers of all involved implementations that vulnerabilities were patched or vulnerable protocols were labeled with appropriate warnings before March 15, 2026 when our results were publicly disclosed. Furthermore, this time window provided the authors of vulnerable protocols an opportunity to update the ePrint versions of their papers if desired, given that they are made public not earlier than the patches of all vulnerable implementations.

We consulted with the maintainers of vulnerable implementations about their fixes. MP-SPDZ is now fixed with a round-wise verification, see §3.4. This also includes the input phase so that an attack similar to that from §5.2 is prevented. The fix for HPMPc implements  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$  (cf. §3.5) for zero-amortized overhead, but uses an alternative instantiation of that based on binary equality circuits with a higher round complexity. Interestingly, we were notified by the maintainer that HPMPc’s implementation of Tetrad [29] is also vulnerable, as minor modifications to the protocol, not contradicting the security proof of Tetrad due to its gap, were made. Finally, ORQ was fixed with our proposed instantiation of  $\mathcal{F}_{\text{CHECKEQS}}^{A, (s_a)_{a \in A}, (t_a)_{a \in A}}$  from §3.5.

**Acknowledgments.** We thank Maximilian Stillger (TU Darmstadt) for our insightful discussions about generalizing and improving formalization of existing security proofs. These discussions are what ultimately lead to the discovery of the flaw in SWIFT from where our research presented in this paper started. We also thank the authors and maintainers of affected papers and implementations for their collaboration in the disclosure process and swiftly patching the affected code.

This project received funding from the ERC under the EU’s research and innovation programs Horizon Europe (PRIVTOOLS/101124778) and Horizon 2020 (PSOTI/850990). It was co-funded by the DFG within SFB 1119 CROSSING/236615297, and supported by BMFTR and HMWK within ATHENE.

**Disclosure of Interests.** The authors have no competing interests to declare.

## References

1. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: ACM CCS 2016, pp. 805–817 (2016). <https://doi.org/10.1145/2976749.2978331>
2. Araki, T., Furukawa, J., Ohara, K., Pinkas, B., Rosemarin, H., Tsuchida, H.: Secure graph analysis at scale. In: ACM CCS 2021, pp. 610–629 (2021). <https://doi.org/10.1145/3460120.3484560>
3. Asharov, G., et al.: Efficient secure three-party sorting with applications to data analysis and heavy hitters. In: ACM CCS 2022, pp. 125–138 (2022). <https://doi.org/10.1145/3548606.3560691>
4. Baum, E., et al.: ORQ: complex analytics on private data with strong security guarantees. In: ACM SOSP 2025, pp. 802–833 (2025). <https://doi.org/10.1145/3731569.3764833>
5. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-69053-0\\_4](https://doi.org/10.1007/3-540-69053-0_4)

6. Boyle, E., Gilboa, N., Ishai, Y., Nof, A.: Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In: ACM CCS 2019, pp. 869–886 (2019). <https://doi.org/10.1145/3319535.3363227>
7. Brüggemann, A., Schick, O., Schneider, T., Suresh, A., Yalame, H.: Don't eject the impostor: fast three-party computation with a known cheater. In: IEEE S&P 2024, pp. 503–522 (2024). <https://doi.org/10.1109/SP54263.2024.00164>
8. Brüggemann, A., Schneider, T.: When trying to catch cheaters breaks the MPC: Breaking and fixing delayed consistency checks in Trident, Fantastic Four, SWIFT, and Quad. Cryptology ePrint Archive, Report 2026/234 (2026). <https://eprint.iacr.org/2026/234>
9. Canetti, R.: Security and composition of multiparty cryptographic protocols. *J. Cryptol.* **13**(1), 143–202 (2000). <https://doi.org/10.1007/s001459910006>
10. Chaudhari, H., Choudhury, A., Patra, A., Suresh, A.: ASTRA: high throughput 3PC over rings with application to secure prediction. In: ACM Cloud Computing Security Workshop (CCSW) 2019, pp. 81–92 (2019). <https://doi.org/10.1145/3338466.3358922>
11. Chaudhari, H., Rachuri, R., Suresh, A.: Trident: Efficient 4PC framework for privacy preserving machine learning. Cryptology ePrint Archive, Report 2019/1315 (2019). <https://eprint.iacr.org/2019/1315>
12. Chaudhari, H., Rachuri, R., Suresh, A.: Trident: efficient 4PC framework for privacy preserving machine learning. In: NDSS 2020 (2020). <https://doi.org/10.14722/ndss.2020.23005>
13. Cohen, R., Lindell, Y.: Fairness versus guaranteed output delivery in secure multiparty computation. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 466–485. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-45608-8\\_25](https://doi.org/10.1007/978-3-662-45608-8_25)
14. Dalskov, A.P.K., Escudero, D., Keller, M.: Fantastic four: honest-majority four-party secure computation with malicious security. In: USENIX Security 2021, pp. 2183–2200 (2021). <https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov>
15. Dalskov, A.P.K., Escudero, D., Nof, A.: Fully secure MPC and zk-FLIOP over rings: new constructions, improvements and extensions. In: CRYPTO 2024, pp. 136–169. LNCS (2024). [https://doi.org/10.1007/978-3-031-68397-8\\_5](https://doi.org/10.1007/978-3-031-68397-8_5)
16. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 572–590. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-74143-5\\_32](https://doi.org/10.1007/978-3-540-74143-5_32)
17. Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 225–255. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-56614-6\\_8](https://doi.org/10.1007/978-3-319-56614-6_8)
18. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press (2004). <https://doi.org/10.1017/CBO9780511721656>
19. Goyal, V., Liu, Y., Song, Y.: Communication-efficient unconditional MPC with guaranteed output delivery. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11693, pp. 85–114. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-26951-7\\_4](https://doi.org/10.1007/978-3-030-26951-7_4)
20. Harth-Kitzerow, C.: HPMPc: High-performance implementation of secure multiparty computation (MPC) protocols. GitHub (2022). <https://github.com/chart21/hmpc/>

21. Harth-Kitzerow, C., Suresh, A., Wang, Y., Yalame, H., Carle, G., Annavaram, M.: High-throughput secure multiparty computation with an honest majority in various network settings. *PoPETs* **2025**(1), 250–272 (2025). <https://doi.org/10.56553/popets-2025-0015>
22. Huang, Y., Katz, J., Evans, D.: Quid-Pro-Quo-tocols: strengthening semi-honest protocols with dual execution. In: *IEEE S&P 2012*, pp. 272–284 (2012). <https://doi.org/10.1109/SP.2012.43>
23. Katz, J., Lindell, Y.: *Introduction to Modern Cryptography*, third edn. Chapman and Hall, CRC Press (2014)
24. Keller, M.: MP-SPDZ: a versatile framework for multi-party computation. In: *ACM CCS 2020*, pp. 1575–1590 (2020). <https://doi.org/10.1145/3372297.3417872>
25. Kiraz, M.S., Schoenmakers, B.: A protocol issue for the malicious case of yao’s garbled circuit construction. In: *27th Symposium on Information Theory in the Benelux*, pp. 283–290 (2006), <https://berry.win.tue.nl/papers/wic06.pdf>
26. Koti, N., Kukkala, V.B., Patra, A., Gopal, B.R.: Graphiti: secure graph computation made more scalable. In: *ACM CCS 2024*, pp. 4017–4031 (2024). <https://doi.org/10.1145/3658644.3670393>
27. Koti, N., Pancholi, M., Patra, A., Suresh, A.: SWIFT: Super-fast and robust privacy-preserving machine learning. *Cryptology ePrint Archive*, Report 2020/592 (2020). <https://eprint.iacr.org/2020/592>
28. Koti, N., Pancholi, M., Patra, A., Suresh, A.: SWIFT: super-fast and robust privacy-preserving machine learning. In: *USENIX Security 2021*, pp. 2651–2668 (2021). <https://www.usenix.org/conference/usenixsecurity21/presentation/koti>
29. Koti, N., Patra, A., Rachuri, R., Suresh, A.: Tetrad: actively secure 4PC for secure training and inference. In: *NDSS 2022* (2022)
30. Mohassel, P., Rindal, P.:  $ABY^3$ : a mixed protocol framework for machine learning. In: *ACM CCS 2018*, pp. 35–52 (2018). <https://doi.org/10.1145/3243734.3243760>