When Trying to Catch Cheaters Breaks the MPC: Breaking and Fixing Delayed Consistency Checks in Trident, Fantastic Four, SWIFT, and Quad (Full Version)*

Andreas Brüggemann[0000−0002−8102−9328]
and Thomas Schneider[0000−0001−8090−1316]

Technical University of 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., [1,31,11,3,22]. Motivating applications are secure computations of large functionalities such as privacy-preserving machine learning [31,11] or analysis of large graphs [2,27]. In

* **Please cite the conference version of this paper, published at EUROCRYPT 2026.** This is the full version of the conference version: [9], ©IACR 2026.

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 [18,13,7,15,29,8,22].

In this work, we investigate the family of protocols from [13,15,29,22], 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 [13,15,29,22] is given in Tab. 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 [15,22] even cite [29] as the basis for their consistency checks. The reuse of the vulnerable construction appears to be ongoing, given the newness of [22] and the same idea appearing in more works such as [30,8] 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 [13]	(§3)	$3+1$	1	Fair	$3+3$	Yes	$\mathcal{O}(1)$
Fantastic Four [15]	(§4)	4	1	Robust	$0+6$	Yes	$\mathcal{O}(1)/\mathcal{O}(d)$[a]
SWIFT [29]	(§5)	3	1	Robust	$3+3$	Depends[b]	–[c]
Quad [22]	(§6)	$3+1$	1	Fair	$2+3$	Yes	$\mathcal{O}(1)$

[a] d: circuit depth; $\mathcal{O}(1)$ for fair protocol variant; $\mathcal{O}(d)$ for robust variant
[b] Protocol underspecified, possible protocol within the specification is insecure
[c] Protocol appears fixable by appropriate, unambiguous specification

Concrete Attacks: Using our generic attack, we discover explicit attacks on the protocols of Trident [13], Fantastic Four [15], SWIFT [29], and Quad [22]. 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 [13], Fantastic Four [15] with fairness, and Quad [22]. 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 [29] by a less ambiguous specification.
Impact on MPC Implementations: Several MPC implementations were vulnerable to our attacks. Fantastic Four [15] is available in MP-SPDZ [25], the most popular code framework for MPC, and directly inherits the flaw of the insecure protocol specification. Furthermore, the recent framework HPMPC [21] pub-

lished together with Quad [22] 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 [25], HPMPC [21], 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 [13,15,29,22]. 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 [6]. 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 [20,16]. This attack uses redundancy in DN-style [17] 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 [20,16] deploy their fix in the protocol's main execution part. For a more detailed comparison to our work, we refer to 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 [26], 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 [23] 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 [13] 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 [15] (§4) while also discussing additional challenges to reach robustness. In §5, we discover that SWIFT's [29] 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 [22] with more details in App. D.

2 Preliminaries

Notation In this work, we mainly consider parties $P_1, \ldots, P_n$. 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 $P_1, \ldots, P_n$ 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 ℓ-bit integers with $\ell \in \mathbb{N}$, or a finite field $\mathbb{F}_{p^k}$ for prime p and $k \in \mathbb{N}$.[1] 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 $[\![x]\!]$, 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 $[\![ax + y + b]\!]$ given arbitrary shares $[\![x]\!], [\![y]\!]$ 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 $[\![x]\!]$ consisting of $x^1, \ldots x^n$, 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 σ and the computational security parameter by κ. Usual parameter choices are $\sigma = 40$ and $\kappa = 128$ which are consistent with [13,15,29,22] 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 κ, 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 [24]. To enable any subset of parties to sample common random values $r \leftarrow\!\!\$\, \mathcal{R}$ non-interactively, we

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

assume that each such subset has established its own independently random and pre-shared PRF key as in [13,15,29,22].

Collision-Resistant Hash Functions The considered works [13,15,29,22] all utilize collision resistant hash functions. This is a family of functions $\mathsf{H}_k : \{0,1\}^* \to \{0,1\}^{\ell(\kappa)}$ ($\ell(\kappa)$ is the output size) where for a key k generated by a probabilistic generator in time polynomial in κ, any efficient adversary with access to k is able to find a collision, i.e., $x \neq x'$ s.t. $\mathsf{H}_k(x) = \mathsf{H}_k(x')$ only with probability negligible in κ. For details and a formal definition, we refer to [24]. Throughout this work, we use H as a hash function without a key as used throughout the protocols in [13,15,29,22]. 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 $|\mathsf{H}|$.

Security Model The protocols from [13,15,29,22] 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 [10] to formalize that an interactive n-party protocol Π securely implements a functionality $\mathcal{F}$. For completeness, we provide details on this security definition in 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) [14,19]. For modularization, we also use the *hybrid model* [10]. Details again are provided in App. A.

3 Our Attack and Fix: Trident [13] as an Example

We begin by providing a detailed explanation of our attack against the four-party protocol Trident [13] 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 [13], 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 [13] 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 Tab. 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 $[\![\cdot]\!]$ where a sharing $[\![v]\!]$ can also be viewed as each party holding m_v and a $\langle\cdot\rangle$-sharing of a random mask λ_v s.t. $\mathsf{m}_v = v + \lambda_v$. Note that the sharings are linear, i.e., given $[\![\cdot]\!]$ respectively $\langle\cdot\rangle$-sharings of $x, y \in \mathcal{R}$ and public $a, b \in \mathcal{R}$, $[\![\cdot]\!]$ 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 $[\![\cdot]\!]$-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 λ_v across the circuit are computed before inputs are provided by the protocol—the online phase only computes the m_v values.

Table 2. Secret sharing semantics for sharing a value $v \in \mathcal{R}$ in Trident [13]. Party P_h holds $\lambda_v^1, \lambda_v^2, \lambda_v^3$ for a sharing $[\![v]\!]$ 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)	$v = v^1 + v^2 + v^3$
$[\![v]\!]$	$(\mathsf{m}_v, \lambda_v^2, \lambda_v^3)$	$(\mathsf{m}_v, \lambda_v^3, \lambda_v^1)$	$(\mathsf{m}_v, \lambda_v^1, \lambda_v^2)$	$\mathsf{m}_v = v + \lambda_v^1 + \lambda_v^2 + \lambda_v^3$

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 $[\![\cdot]\!]$-sharings. Finally, for input and output phases, we simply refer to [13] as they play no role for our attack. To multiply two sharings $[\![x]\!], [\![y]\!]$, observe the following using the linearity of $\langle\cdot\rangle$-sharings:

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

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 [13], we abstract it away into a functionality $\mathcal{F}_{\text{MultPre}}$ (as also used in, e.g., [29]), 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, $\langle x \cdot y \rangle = \mathsf{m}_x \cdot \mathsf{m}_y - \mathsf{m}_x \cdot \langle \lambda_y \rangle - \mathsf{m}_y \cdot \langle \lambda_x \rangle + \langle \gamma_{xy} \rangle$ can be computed non-interactively by the linearity of $\langle\cdot\rangle$-sharings. The goal now is to obtain a sharing $[\![z]\!]$ 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 $\mathsf{m}'_z := \mathsf{m}_z - \mathsf{m}_x \mathsf{m}_y$ from where they can non-interactively add $\mathsf{m}_x \mathsf{m}_y$ to finally obtain m_z and, hence, the output sharing $[\![z]\!]$.

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

Now, the resulting sharing $\langle \mathsf{m}'_z \rangle$ is opened interactively to obtain 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 $\langle \mathsf{m}'_z \rangle$ 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 m'^i_z while the other provides $\mathsf{H}(\mathsf{m}'^i_z)$. In case of an incorrect m'^i_z 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 $\Pi_{\mathsf{mult}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \to \llbracket z \rrbracket$ of Trident [13]

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:
   - $\mathsf{m}_z'^{i-1} = -\mathsf{m}_x \lambda_y^{i-1} - \mathsf{m}_y \lambda_x^{i-1} + \gamma_{xy}^{i-1} + \lambda_z^{i-1}$.
   - $\mathsf{m}_z'^{i+1} = -\mathsf{m}_x \lambda_y^{i+1} - \mathsf{m}_y \lambda_x^{i+1} + \gamma_{xy}^{i+1} + \lambda_z^{i+1}$.
2. Each party P_i sends $\mathsf{m}_z'^{i-1}$ to P_{i-1}.
3. Each party P_i locally computes $\mathsf{m}_z = \mathsf{m}_x \mathsf{m}_y + \sum_{j=1}^{3} \mathsf{m}_z'^{j}$.

Verify:

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

Fig. 1. Multiplication protocol of Trident [13] (simplified preprocessing).

3.1 Security of Unoptimized Trident

The security proof of Trident is deferred to the full version [12] of the conference version [13]. Important here is how the simulation of the online and verification phases of multiplications works. According to [12], 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 $\mathsf{m}_z'^{i}$ received by a corrupt P_i in the online phase of a multiplication is simulated by handing $\mathsf{m}_z'^{i}$ (from the emulation of P_{i+1}) to the adversary on behalf of P_{i+1}. Regarding verification, the simulator hands $\mathsf{H}(\mathsf{m}_z'^{i})$ 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 $\mathsf{m}_z'^{i}$ received by the adversary cannot be distinguished from a real-world execution of the same step. This is because the value is masked by λ_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 $\mathsf{H}(\mathsf{m}_z'^i)$ 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 $\mathsf{m}_z'^i$. 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 [12,13], an oversight that enables the issues discussed in the following. Yet, the entire proof in [12] appears to implicitly consider an immediate verification, not matching the optimized protocol description in [12,13], 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 $[\![u]\!], [\![v]\!], [\![w]\!]$. Then, P_i receives $\mathsf{m}_u'^i, \mathsf{m}_v'^i, \mathsf{m}_w'^i$ from P_{i+1} which can be verified by P_{i-1} sending a single hash (instead of three separate ones) $h_i = \mathsf{H}(\mathsf{m}_u'^i \| \mathsf{m}_v'^i \| \mathsf{m}_w'^i)$ 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 Π_{mult} (Fig. 1) and the batched verification, jointly depicted in Fig. 3.

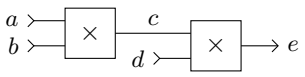


Fig. 2. Simple arithmetic circuit featuring two subsequent multiplications, providing a minimal example to enable our attack in §3.3.

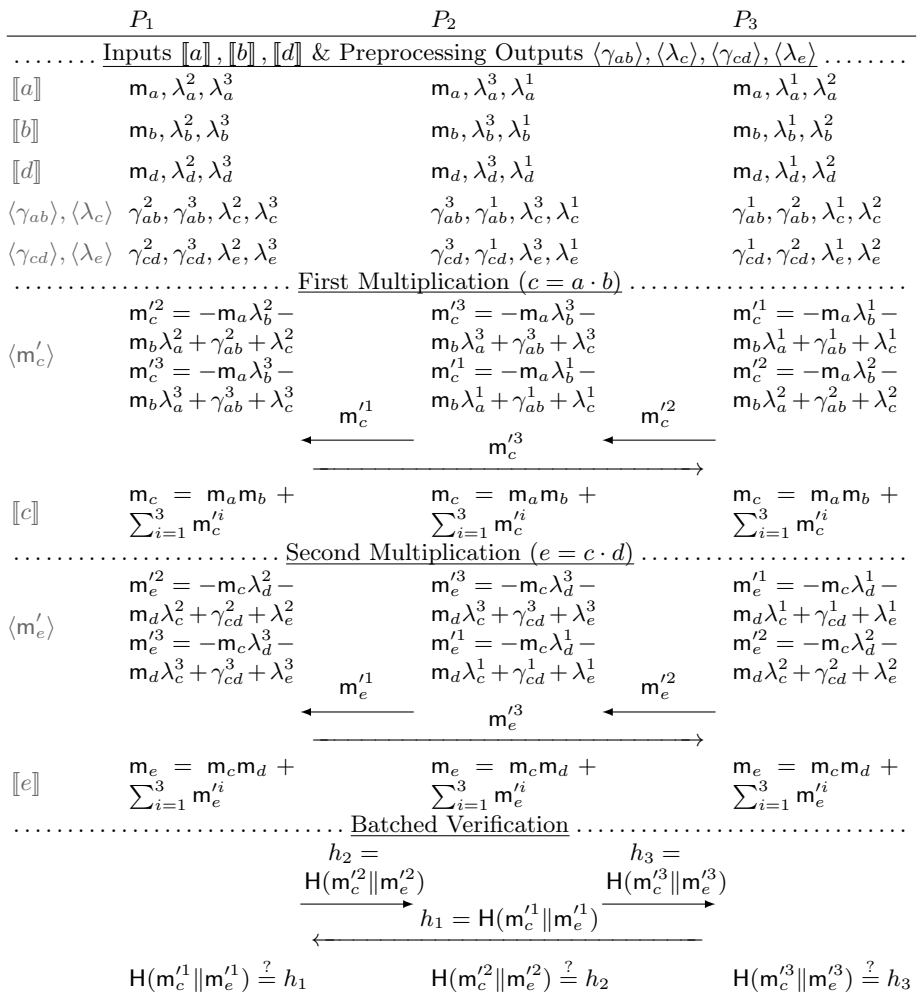


Fig. 3. Online phase of the two multiplications and batched verification required to evaluate the circuit from Fig. 2 in Trident [13], following the notation from Fig. 1.

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 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 $\mathsf{m}_c'^3$ to P_3. Our attack is to instead incorrectly send $\mathsf{m}_c'^3 + 1$. This will cause P_3 to eventually recover $\mathsf{m}_c + 1$ instead of the correct value m_c. This error propagates to further values in the second multiplication. In particular, P_3 has to compute $\mathsf{m}_e'^1 = -\mathsf{m}_c\lambda_d^1 - \mathsf{m}_d\lambda_c^1 + \gamma_{cd}^1 + \lambda_e^1$, but the "+1"-offset on m_c causes it to instead compute $\mathsf{m}_e'^1 - \lambda_d^1$. The error has not propagated to P_2 yet which computes the correct $\mathsf{m}_e'^1$ and sends it to P_1. In the verification, P_3 receives $h_3 = \mathsf{H}(\mathsf{m}_c'^3 \| \dots)$ from P_2 which except for negligible probability is unequal to $\mathsf{H}(\mathsf{m}_c'^3 + 1 \| \dots)$. Hence, the cheating attempt is caught as soundness still holds. Yet, the corrupt party P_1 receives from P_3 $h_1 = \mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 - \lambda_d^1)$, not necessarily matching $\mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1)$ 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 [12] has a value λ_d^1, yet it uses zero-inputs for all honest parties, relying on the adversary not knowing the λ_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 $h_1 = \mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 - \delta)$ to the adversary where δ may depend only on what the simulator knows and, hence, not on value d. In a real execution with $d = 0$, the adversary receives $h_1 = \mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 - \lambda_d^1) = \mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 - \mathsf{m}_d + \lambda_d^2 + \lambda_d^3)$ and already has $\mathsf{m}_c'^1, \mathsf{m}_e'^1, \lambda_d^2, \lambda_d^3$. For $d = 1$, the adversary receives $\mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 - \lambda_d^1) = \mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 + 1 - \mathsf{m}_d + \lambda_d^2 + \lambda_d^3)$. Hence, the distributions $\left\{h_1, \mathsf{m}_c'^1, \mathsf{m}_e'^1, \lambda_d^2, \lambda_d^3\right\}$ 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 $h_1 = \mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 - \delta)$ for δ *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. $\mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 - i) = h_1$. Except for the unlikely case

of a hash collision, it then holds that $i = \lambda_d^1$ and P_1 that already has $\mathsf{m}_d, \lambda_d^2, \lambda_d^3$ can extract the plaintext $d = \mathsf{m}_d - \lambda_d^1 - \lambda_d^2 - \lambda_d^3$ 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 $\mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 + x - \mathsf{m}_d + \lambda_d^2 + \lambda_d^3) = h_1$, i.e., if $x - \mathsf{m}_d + \lambda_d^2 + \lambda_d^3 = \lambda_d^1 \Leftrightarrow d = x$. 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 $h_1 = \mathsf{H}(\dots \| \mathsf{m}_c'^1 \| \dots \| \mathsf{m}_e'^1 - \lambda_d^1 \| \dots)$ to compare to $\mathsf{H}(\dots \| \mathsf{m}_c'^1 \| \dots \| \mathsf{m}_e'^1 \| \dots)$ with the omitted parts being consistent. The attack works as for $h_1 = \mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1 - \lambda_d^1)$ to compare to $\mathsf{H}(\mathsf{m}_c'^1 \| \mathsf{m}_e'^1)$ 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 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 [13]. As all instances of $\mathcal{F}_{\text{MultPre}}$ are used in parallel by [13], 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 [23]. 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 wether $\lambda_d^1 = 0$ which would imply that $d = \mathsf{m}_d + \lambda_d^2 + \lambda_d^3$, 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: $h_a^1, h_a^2 \in \{0,1\}^{|\mathsf{H}|}$ 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 $\Pi_{\mathsf{mult}}^{\text{FIXED}}$. The "..." here stands for the respective $\mathsf{m}_z'^i, \mathsf{m}_z'^{i+1}$ 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 $\Pi_{\mathsf{mult}}^{\text{FIXED}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \to \llbracket z \rrbracket$ for Trident [13]

Verify (batched):

1. Each P_i sends $h_i^1 = \mathsf{H}(\dots \| \mathsf{m}_z'^i \| \dots)$ and $h_{i+1}^2 = \mathsf{H}(\dots \| \mathsf{m}_z'^{i+1} \| \dots)$ 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 [13] 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 [13] using the fixed multiplication protocol $\Pi_{\mathsf{mult}}^{\text{FIXED}}$ (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 [12] 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 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 [12]. Also, it is easy to see that the protocol implements the desired functionality. Inputs are provided through the protocols in [13], where $\mathcal{S}$ lets honest parties share 0 instead of their actual

inputs and can extract the corrupt party's inputs as described in [12]. It is crucial here that consistency checks used internally by the input protocol, as per the protocol description in [13], are not delayed. Otherwise, the input protocol could again be used to cause inconsistencies, which may break privacy as we observe in the case of SWIFT [29] in §5.2.

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'^{i}_z to $\mathcal{A}$ on behalf of P_{i+1}. Note that this message is masked by λ^i_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'^{i-1}_z (not necessarily consistent with m'^{i-1}_z 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'^{i-1}_z 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 $h^1_i, h^2_{i+1} \in \{0,1\} \in \{0,1\}^{|\mathsf{H}|}$ that P_i is supposed to send to $\mathcal{F}^{\text{Trident}}_{\text{CheckEqs}}$. If it does not receive these values, it simulates an abort as above. Then, $h^1_{i-1}, h^1_{i+1}, h^2_{i-1}, h^2_i$ are computed by $\mathcal{S}$ on the shares of their respective honest parties. If $h^1_j \neq h^2_j$ for any $j \in \{1,2,3\}$, then $\mathcal{S}$ hands $\texttt{Reject}$ to $\mathcal{A}$ on behalf of $\mathcal{F}^{\text{Trident}}_{\text{CheckEqs}}$, sends abort to $\mathcal{F}$, simulates the other parties aborting and outputs what $\mathcal{A}$ outputs. Otherwise, it sends $\texttt{Accept}$ to $\mathcal{A}$ on behalf of $\mathcal{F}^{\text{Trident}}_{\text{CheckEqs}}$ 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 $[\![z]\!]$ by sending some $\mathsf{m}'^{i-1}_z + \delta, \delta \in \mathcal{R}, \delta \neq 0$ instead of m'^{i-1}_z. 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'^{i-1}_z. Now, P_{i+1} includes m'^{i-1}_z in h^2_{i-1} while P_{i-1} includes the received $\mathsf{m}'^{i-1}_z + \delta$ in h^1_{i-1}. Then, except for negligible probability of a hash collision, $h^1_{i-1} \neq h^2_{i-1}$ so that in the ideal and in the real world, the output of $\mathcal{F}^{\text{Trident}}_{\text{CheckEqs}}$ will be $\texttt{Reject}$.

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

The output phase is simulated as in [12] and $\mathcal{S}$ outputs what $\mathcal{A}$ outputs. □

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

away to functionality $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$, evaluating an arithmetic circuit $\mathcal{C}$ on inputs provided by n parties and, in addition, k uniform and secret random values:

$$\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}(\mathcal{C}, x_{1,1}, \ldots, x_{1,j_1}, \ldots, x_{n,1}, \ldots, x_{n,j_n}) = \\ \mathcal{C}(x_{1,1}, \ldots, x_{1,j_1}, \ldots, x_{n,1}, \ldots, x_{n,j_n}, r_1 \leftarrow\!\!\$\, \mathbb{F}_{2^{|\mathsf{H}|}}, \ldots, r_k \leftarrow\!\!\$\, \mathbb{F}_{2^{|\mathsf{H}|}}),$$

with arithmetic circuit $\mathcal{C}$, inputs $x_{i,1}, \ldots, x_{i,j_i} \in \mathbb{F}_{2^{|\mathsf{H}|}}$ 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 $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$ 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 $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$. 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 $\mathbb{F}_{2^{|\mathsf{H}|}}$. Note that this field must be and indeed is large, s.t. soundness is violated with probability negligible in the statistical security parameter σ. We then compute a random linear combination using $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$ 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: $h_a^1, h_a^2 \in \{0,1\}^{|\mathsf{H}|}$ (interpreted as elements of $\mathbb{F}_{2^{|\mathsf{H}|}}$) 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 secret $r_a \leftarrow\!\!\$\, \mathbb{F}_{2^{|\mathsf{H}|}}, a \in A$, using $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$.
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 $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$-hybrid model, assuming that $|\mathsf{H}| > \sigma$.*

Proof. For value x computed by $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$, 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 $\mathbb{F}_{2^{|\mathsf{H}|}}$, independently of all other r_a and h values. Then, the probability of it satisfying the prior equation and hence accepting is $|\mathbb{F}_{2^{|\mathsf{H}|}}|^{-1} = 2^{-|\mathsf{H}|} \leq 2^{-\sigma}$ and, hence, negligible. Thus, the protocol correctly instantiates the desired functionality.

Regarding simulation, $\mathcal{S}$ receives from $\mathcal{A}$ all of its inputs to $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$ or, if no or insufficient input is provided, sends `abort` to $\mathcal{F}_{\textsc{CheckEqs}}^{A,(s_a)_{a\in A},(t_a)_{a\in A}}$, simulates the other parties aborting, and outputs what $\mathcal{A}$ outputs. It sends $\mathcal{A}$'s inputs to $\mathcal{F}_{\textsc{CheckEqs}}^{A,(s_a)_{a\in A},(t_a)_{a\in A}}$, receiving back either `Accept` or `Reject`. If $\mathcal{S}$ receives `Accept`, then it provides 0 to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$. Otherwise, it samples uniformly random $x' \in \mathbb{F}_{2^{|\mathsf{H}|}}$ and provides that to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$. 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 $\mathcal{F}_{\textsc{CheckEqs}}^{A,(s_a)_{a\in A},(t_a)_{a\in A}}$ 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 $\mathbb{F}_{2^{|\mathsf{H}|}}$. As $r_{\hat{a}}$ is uniformly random, so is x, proving that x and x' cannot be distinguished. □

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

$\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$ can be instantiated in this setting using any secure MPC protocol for the same setting that supports arithmetic circuits over $\mathbb{F}_{2^{|\mathsf{H}|}}$ 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 $[\![r]\!]$ 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 $\mathsf{m}_r = 0$. 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 $\mathcal{R} = \mathbb{F}_{2^{|\mathsf{H}|}}$. 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 $\Pi_{\textsc{CheckEqs}}^{A,(s_a)_{a\in A},(t_a)_{a\in A}}$ has a constant multiplicative depth of only one, yielding minimal round overhead. More concretely, the immediate verification within $\Pi_{\textsc{CheckEqs}}^{\textsc{Trident}}$ 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.[2] Thus, the concrete overhead is negligible and the use of $\mathcal{R} = \mathbb{F}_{2^{|\mathsf{H}|}}$ during verification comes at a negligible performance penalty.

[2] As another optimization, [13] also provides an efficient protocol to compute a dot product at the cost of a single scalar multiplication. Given that the random linear combination to be computed can be interpreted as a dot product, the three multiplications and two additions can be replaced by a single dot product to further decrease communication cost. The same also holds for other protocols such as [15,29,22].

4 Attacking and Fixing Fantastic Four [15]

We proceed to extend our attack on Trident [13] from §3 to the robust four-party protocol from Fantastic Four [15].[3] The attack is enabled by the protocol using the same idea for delayed, batched consistency checks as Trident [13]. 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 $[\![v]\!]$ and note that it is linear.

The intuition to multiply $[\![x]\!], [\![y]\!]$ 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.

Protocol $\Pi_{\mathsf{mult}}([\![x]\!], [\![y]\!]) \to [\![z]\!]$ of Fantastic Four [15]

Input: $[\![\cdot]\!]$-shares of $x, y \in \mathcal{R}$.

Output: $[\![\cdot]\!]$-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 $[\![z_{gh}]\!]$ with $z_{gh} = x^g y^h + x^h y^g$.
   - P_i, P_j, P_h non-interactively sample $z^g_{gh} \leftarrow\!\!\$\ \mathcal{R}$.
   - P_i, P_j compute $z^h_{gh} = x^g y^h + x^h y^g - z^g_{gh}$.
   - P_i sends z^h_{gh} to P_g.
   - Set $z^i_{gh} = z^j_{gh} = 0$.
2. Each party P_i sets $z^j = x^j y^j + \sum_{\{g,h\} \subseteq \{1,2,3,4\}, g \neq h} z^j_{gh}$ 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^h_{gh} to P_g:
   - P_j sends $h_{gh} = \mathsf{H}(z^h_{gh})$ to P_g.
   - Party P_g proceeds if $h_{gh} = \mathsf{H}(z^h_{gh})$ (with z^h_{gh} as previously received from P_i).
   - In case of inequality, output error message to other parties and start cheater identification as described in [15].

Fig. 7. Four-party multiplication protocol of Fantastic Four [15] (some operations inlined and separated verification).

Fantastic Four batches the verification at the end of optimistically executed segments which are not explicitly defined in [15]. Yet, in the implementation of

[3] For simplicity, we call the protocol "Fantastic Four" here while noting that [15] also contains a three-party protocol that is not relevant to this work.

the protocol in MP-SPDZ [25] (`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^h_{gh} to P_g and who later sends the hash in the verification phase. While [15] 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 [25]. We observe that the implementation does not strictly adhere to the $g < h$ requirement of [15] (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 Tab. 3.

Table 3. Assignment of the parties' roles in the MP-SPDZ [25] (`v0.4.2`) implementation of Fantastic Four [15].

$\{g, h\}$	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

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 $c^1_{12} + 1$ instead of $c^1_{12} = a^2 b^1 + a^1 b^2 - c^2_{12}$, where $c^2_{12} \leftarrow\!\!\$\ \mathcal{R}$. 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 $e^1_{13}, e^4_{14}, e^4_{34}$, only the last being independent of incorrect $c^1 + 1$. For the other values, the offset on c^1 causes it to compute

$$c^3 d^1 + (c^1 + 1) d^3 - e^3_{13} = e^1_{13} + d^3 \text{ instead of } e^1_{13}, \text{ and}$$
$$(c^1 + 1) d^4 + c^4 d^1 - e^1_{14} = e^4_{14} + d^4 \text{ instead of } e^4_{14}.$$

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

It is easy to see that the consistency check will fail as P_2 receives $h_{12} = \mathsf{H}(c^1_{12} \| e^1_{12})$ from P_4, which except for negligible probability does not match $\mathsf{H}(c^1_{12} + 1 \| e^1_{12})$ for the values received from P_3. Yet, we observe that the corrupt party P_3 receives c^1_{13} and $e^1_{13} + d^3$ from P_2 while it receives $h_{13} = \mathsf{H}(c^1_{13} \| e^1_{13})$

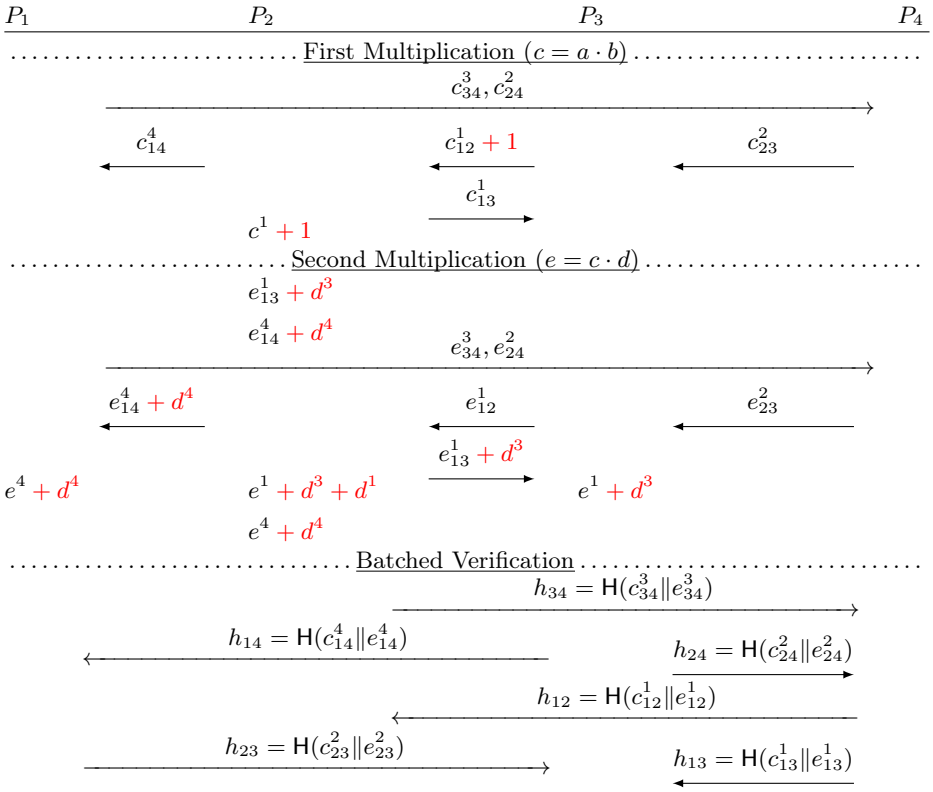


Fig. 8. Two multiplications and batched verification required to evaluate the circuit from Fig. 2 in Fantastic Four [15]. Parts in red are errors introduced by cheating P_3, and we omit all locally held shares that do not contain any error. We also omit the comparison of hash values and cheater identification inside the batched verification, as they play no role in our attack.

from P_4. Hence, similar to the attack in §3.3, P_3 may for example check for different $x \in \mathcal{R}$ if $\mathsf{H}(c^1_{13} \| e^1_{13} + d^3 - x) = h_{13}$. 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 [25]. 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 $\Pi_{\mathsf{mult}}^{\text{FIXED}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \to \llbracket z \rrbracket$ for Fantastic Four [15]

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_g:
   - P_j sends $h_{gh}^1 = \mathsf{H}(\dots \| z_{gh}^h \| \dots)$ to $\mathcal{F}_{\text{CheckEqs}}^{\text{FantasticFour}}$.
   - P_g sends $h_{gh}^2 = \mathsf{H}(\dots \| z_{gh}^h \| \dots)$ (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 [15] in Fig. 7.

To instantiate $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$ 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 [15] 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 [15] 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 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 [29]: Almost-Evasion of our Attack

The three-party protocol of SWIFT [29][4] is another protocol using delayed verification checks and targeting robustness. The core idea (regarding multiplication) of SWIFT is very close to that of Trident [13] (§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

[4] For simplicity, we call the protocol "SWIFT" here, noting that [29] also contains a four-party protocol that we do not examine in this paper.

in §4.2 for Fantastic Four [15]. Yet, we note that the close similarity to Trident is present only in more recent rephrased and simplified versions of SWIFT, especially that in [8]. We decide here to target the original protocol in [29], noting that more recent versions are not consistent with each other and do not claim to resolve any security issue. For consistency with [29], 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 Tab. 4. The main sharing semantic is $[\![\cdot]\!]$ 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 β_v-values are computed online.

Table 4. Secret sharing semantics for sharing a value $v \in \mathcal{R}$ in SWIFT [29].

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

The multiplication protocol of SWIFT is depicted in Fig. 10. It already includes optimizations that [29] only describes in text. Importantly, [29] 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.

Protocol $\Pi_{\mathsf{mult}}([\![x]\!], [\![y]\!]) \to [\![z]\!]$ of SWIFT [29]

Input: $[\![\cdot]\!]$-shares of $x, y \in \mathcal{R}$.

Output: $[\![\cdot]\!]$-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 [29] for details):

– $\langle\chi\rangle$ where in addition, P_0 has both shares χ^1, χ^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 $h_0 = \mathsf{H}(\beta_z + \gamma_z)$ to P_0.
2. If P_0 detects that $h_0 \neq \mathsf{H}(\beta_z + \gamma_z)$ with $\beta_z + \gamma_z$ as received by P_1, it outputs an error message and starts the cheater identification described in [29]. 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 $h_i = \mathsf{H}(\beta_z^{*\,3-i})$ to P_i for $i \in \{1,2\}$.
5. If P_i for $i \in \{1,2\}$ detects that $h_i \neq \mathsf{H}(\beta_z^{*\,3-i})$ with $\beta_z^{*\,3-i}$ as received by P_{3-i}, it outputs an error message and starts the cheater identification described in [29].

Fig. 10. Multiplication protocol of SWIFT [29] (some operations inlined and restructured according to the optimizations that [29] only describes in text).

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 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 β_x, β_y that they would hold in an honest protocol execution. Now, in a multiplication, P_1 uses $\hat{\beta}_z^{*\,1} + \delta = -(\hat{\beta}_x + \gamma_x)\alpha_y^1 - (\hat{\beta}_y + \gamma_y)\alpha_x^1 + \alpha_z^1 + \chi^1 + \delta$ 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 δ. 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 $\hat{\beta}_c = \beta_c + \delta$.[5] In

[5] Of course, from this, P_1 can also derive β_c given that it selects the error δ.

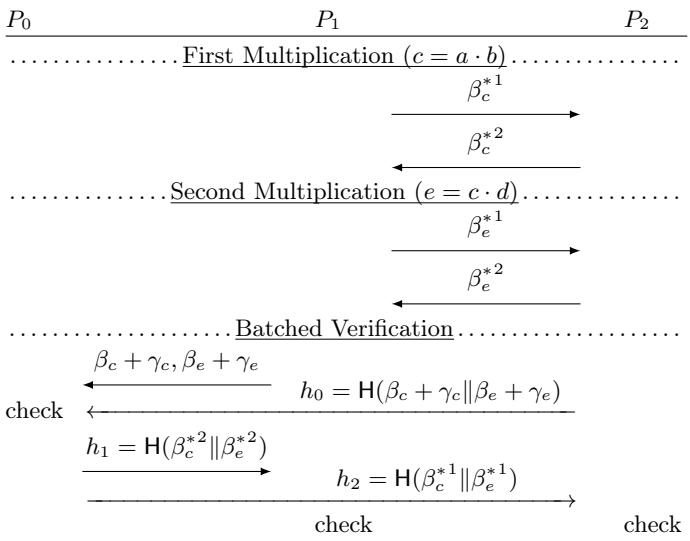


Fig. 11. Online phase of the two multiplications and batched verification required to evaluate the circuit from Fig. 2 in SWIFT [29].

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 α_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 β_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 [29] nor in the full version [28] that it refers to for the security proof. Instead, like for Trident [13] as discussed in §3.1, it appears that [28] 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 [29] 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 $\mathcal{F}_{\text{CheckEqs}}^{A,(s_a)_{a\in A},(t_a)_{a\in A}}$ 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 [29], 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 β_v to P_1.
5. P_0, P_1 "jmp-send β_v to P_2" [29, p. 2656].

The "jmp-send" operation denotes that one of the parties P_0, P_1 sends β_v and the other sends $\mathsf{H}(\beta_v)$ to P_2 that then can check for consistency in [29]. While the distribution of roles is ambiguous in the paper, the notation [29, Notation 3.1 and Fig. 1] suggests that the first mentioned party, i.e., P_0 sends the actual value β_v. It is the same primitive that [29] 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" [29, Sect. 3.1] and "while the *verify* for a fixed ordered pair of senders will be executed once and for all in the end" [29, Sect. 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, [29] 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 β_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 $h_0 = \mathsf{H}(\beta_c + \beta_b - \alpha_b^2 + \gamma_c)$. As in §3.3, this enables P_0 to test for certain or even fully extract β_b while it already has α_b^1, α_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.

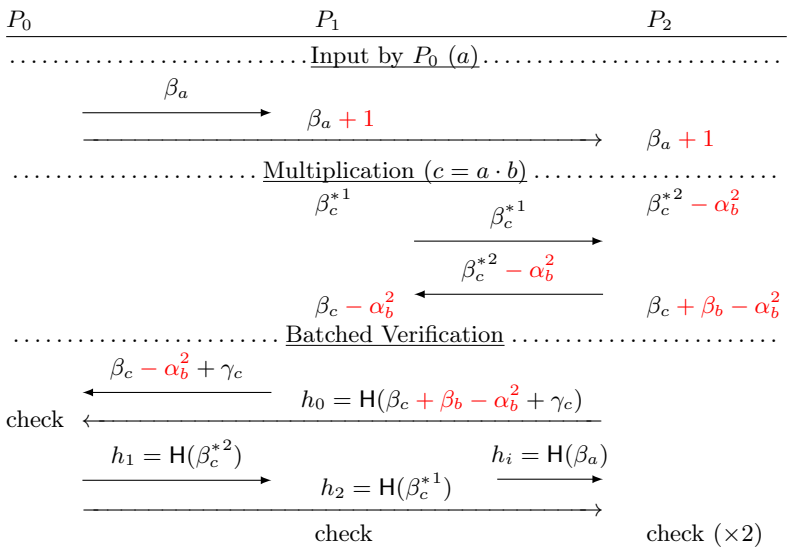


Fig. 12. One input (other correct and omitted), one multiplication and batched verification required to evaluate $c = a \cdot b$ in SWIFT [29]. Parts in red are errors introduced by cheating P_0 and we omit all locally held shares that do not contain any error.

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 [30] and SOCIUM [8] The four-party protocol Tetrad [30] improves upon Trident [13] with the online phase of multiplications deviating from Tetrad mostly by using the same trick as SWIFT [29] 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 [8] 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 [22]

The last instance of our attack is on the fair four-party protocol Quad [22]. We note that the underlying ideas of Quad are quite similar to those in Trident [13] (discussed in §3), also using a sharing semantic $[\![v]\!]$ consisting of a masked value $\mathsf{m}_v = v + \lambda_v$ and a mask λ_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 App. D.

7 Responsible Disclosure

We found the vulnerable protocols to be implemented in the MPC frameworks MP-SPDZ[6] [25], HPMPC[7] [21,22], and in the oblivious analytics system Orq[8] [4]. The frameworks inherited the vulnerabilities from the insecure protocols at the

[6] https://github.com/data61/MP-SPDZ/, `v0.4.2`: Fantastic Four [15]
[7] https://github.com/chart21/hpmpc/, `a9809dc`: Fantastic Four [15] and Quad [22]
[8] https://github.com/CASP-Systems-BU/orq, `v1.0.0`: Fantastic Four [15]

time of disclosure. After acceptance of our paper, we informed the authors of all affected protocols and implementations on January 30, 2026. We coordinated with the maintainers of all involved implementations that vulnerabilities were patched or vulnerable protocol implementations were labeled with appropriate warnings before March 15, 2026. Furthermore, this time window provided the authors of vulnerable protocols an opportunity to update the ePrint versions of their papers if desired.

We consulted with the maintainers of vulnerable implementations about their fixes. MP-SPDZ is now fixed (commit 85307a1, included in release `v0.4.3`) 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 (finalized in commit d74efd0 on branch `dev`) 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 [30] was also vulnerable, as minor modifications to the protocol, not contradicting the security proof of Tetrad due to its gap, were made. Finally, Orq has been fixed in version `v2.0.0`, using 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. Finally, we thank Eli Baum (Boston University) for pointing out that the instantiation of $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{sub}}$ for our fix can be optimized further using specialized dot product protocols which are available in all of the protocols considered in this work.

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., Hamada, K., Ikarashi, D., Kikuchi, R., Nof, A., Pinkas, B., Takahashi, K., Tomida, J.: 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., Buxbaum, S., Mathai, N., Faisal, M., Kalavri, V., Varia, M., Liagouris, J.: ORQ: Complex analytics on private data with strong security guarantees. In: ACM SOSP 2025. p. 802–833 (2025). https://doi.org/10.1145/3731569.3764833
5. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-knowledge proofs on secret-shared data via fully linear PCPs. In: CRYPTO 2019. pp. 67–97. LNCS (2019). https://doi.org/10.1007/978-3-030-26954-8_3
6. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults (extended abstract). In: EUROCRYPT 1997. pp. 37–51. LNCS (1997). https://doi.org/10.1007/3-540-69053-0_4
7. 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
8. 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
9. 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. In: EUROCRYPT 2026. pp. 457–487. LNCS (2026). https://doi.org/10.1007/978-3-032-25324-8_16
10. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology **13**(1), 143–202 (2000). https://doi.org/10.1007/s001459910006
11. 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. p. 81–92 (2019). https://doi.org/10.1145/3338466.3358922
12. 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
13. 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
14. Cohen, R., Lindell, Y.: Fairness versus guaranteed output delivery in secure multiparty computation. In: ASIACRYPT 2014. pp. 466–485. LNCS (2014). https://doi.org/10.1007/978-3-662-45608-8_25
15. 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
16. 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
17. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: CRYPTO 2007. pp. 572–590. LNCS (2007). https://doi.org/10.1007/978-3-540-74143-5_32
18. Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: EUROCRYPT 2017. pp. 225–255. LNCS (2017). https://doi.org/10.1007/978-3-319-56614-6_8
19. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press (2004). https://doi.org/10.1017/CBO9780511721656

20. Goyal, V., Liu, Y., Song, Y.: Communication-efficient unconditional MPC with guaranteed output delivery. In: CRYPTO 2019. pp. 85–114. LNCS (2019). https://doi.org/10.1007/978-3-030-26951-7_4
21. Harth-Kitzerow, C.: HPMPC: High-performance implementation of secure multiparty computation (MPC) protocols. GitHub: https://github.com/chart21/hpmpc/ (2022)
22. 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
23. 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
24. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall, CRC Press, third edn. (2014)
25. 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
26. 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
27. 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
28. 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
29. 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
30. Koti, N., Patra, A., Rachuri, R., Suresh, A.: Tetrad: Actively secure 4PC for secure training and inference. In: NDSS 2022 (2022)
31. 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

A Details on the Security Model

Throughout this paper, we use the standard simulation-based real-world/ideal-world security definition [10] assuming an active, nonadaptive adversary corrupting one party and the existence of pairwise secure channels between the parties. For the sake of completeness, we summarize this definition in the following:

We denote a functionality mapping n inputs to n outputs, one input and output per party by $\mathcal{F}$. Interactive protocol Π between the n parties then securely implements $\mathcal{F}$ if the following holds:

In the *real world*, $\mathcal{A}$ receives the identity of the corrupted party and its private input, together with auxiliary input z, the security parameter κ, and a random tape r. Then, following the specification of the protocol, for each communication round, the uncorrupted parties compute their messages to send and exchange them, providing $\mathcal{A}$ with all messages directed at the corrupt party.

Adversary $\mathcal{A}$ then determines the messages to be sent by the corrupted party, and the round is concluded. At the end of the protocol, all uncorrupted parties compute and output their protocol outputs according to the protocol definition. The corrupted party instead outputs a special symbol, and $\mathcal{A}$ provides an arbitrary output, depending on what it learned from the protocol execution and its inputs. Then, $\text{EXEC}_{\Pi,\mathcal{A}}(\kappa, \mathbf{x}, z)$ is the probability distribution over the concatenation over the outputs of $\mathcal{A}$ and all parties, where $\mathbf{x}$ contains the inputs of all parties for uniformly random sampling of random tapes for all parties and $\mathcal{A}$. Furthermore, $\text{EXEC}_{\Pi,\mathcal{A}} := \{\text{EXEC}_{\Pi,\mathcal{A}}(\kappa, \mathbf{x}, z)\}_{\kappa,\mathbf{x},z}$.

In the *ideal world*, adversary $\mathcal{S}$ receives the identity of the corrupted party and its private input, together with auxiliary input z, the security parameter κ, and a random tape r. The ideal functionality $\mathcal{F}$ is then computed on the honest parties' inputs and an arbitrary input provided by $\mathcal{S}$ replacing the corrupt party's input. Each uncorrupted party outputs its output from the prior evaluation of $\mathcal{F}$, the corrupted party outputs a special symbol, and $\mathcal{S}$ provides an arbitrary output, depending on what it learned from the output of $\mathcal{F}$ dedicated to the corrupt party and its inputs. Then, $\text{IDEAL}_{\mathcal{F},\mathcal{S}}(\kappa, \mathbf{x}, z)$ is the probability distribution over the concatenation over the outputs of $\mathcal{S}$ and all parties, otherwise defined as $\text{EXEC}_{\Pi,\mathcal{A}}(\kappa, \mathbf{x}, z)$ before. Furthermore, $\text{IDEAL}_{\mathcal{F},\mathcal{S}} := \{\text{IDEAL}_{\mathcal{F},\mathcal{S}}(\kappa, \mathbf{x}, z)\}_{\kappa,\mathbf{x},z}$.

We say that Π securely implements $\mathcal{F}$ if for each real-world adversary $\mathcal{A}$ there exists an efficient ideal-world adversary $\mathcal{S}$, called simulator, such that $\text{EXEC}_{\Pi,\mathcal{A}}$ and $\text{IDEAL}_{\mathcal{F},\mathcal{S}}$ are computationally indistinguishable.

The ideal functionality $\mathcal{F}$ may be augmented to provide an adversary with some additional powers, yielding more specific security models as discussed in, e.g., [14,19] which we briefly introduce: **Robustness/Guaranteed Output Delivery** guarantees that all parties receive a result when executing Π. Note that in the ideal world, $\mathcal{S}$ still has the opportunity to provide no inputs. Hence, $\mathcal{F}$ is extended by default values for when a party provides no inputs. **Fairness** allows an adversary to abort the computation, but only s.t. either all or no parties receive their outputs. To this end, $\mathcal{S}$ is allowed to send an abort signal to $\mathcal{F}$ as the corrupted party's input, leading to all parties receiving an abort instead of their output. Finally, for **Security with Abort**, the computation may be aborted as before, but in addition $\mathcal{F}$ will first provide outputs to the corrupt party and then allows $\mathcal{S}$ to decide if the other parties receive an abort or their output.

We also use the *hybrid model* [10], where access to a trusted party computing certain sub-functionalities exists. If a protocol Π is secure in such a hybrid model and there are secure protocols for each of the sub-functionalities, then Π', which replaces each invocation of an ideal sub-functionality inside Π by the respective secure protocol, is also secure [10].

B Attack on Trident [13] and a More Complex Circuit

Here we provide an example of our attack on Trident [13] from §3.3 in case of a more complex circuit to evaluate. Our example circuit for that is provided in Fig. 13. As in §3.3, this could be extended arbitrarily as long as the output of the first multiplication does not influence any new multiplications.

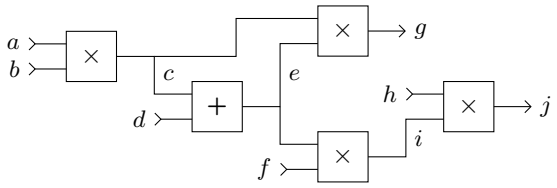


Fig. 13. Example for a more complex arithmetic circuit that is affected by our attack.

We let corrupt P_1 again send offset $\mathsf{m}_c'^3 + 1$ to P_3 in the first multiplication, but internally use $\mathsf{m}_c'^3$ to compute m_c. Recall that for a multiplication $z = xy$, each party P_j computes

$$\mathsf{m}_z'^{j-1} = -\mathsf{m}_x\lambda_y^{j-1} - \mathsf{m}_y\lambda_x^{j-1} + \gamma_{xy}^{j-1} + \lambda_z^{j-1},$$
$$\mathsf{m}_z'^{j+1} = -\mathsf{m}_x\lambda_y^{j+1} - \mathsf{m}_y\lambda_x^{j+1} + \gamma_{xy}^{j+1} + \lambda_z^{j+1},$$

sends $\mathsf{m}_z'^{j-1}$ to P_{j-1}, receives $\mathsf{m}_z'^{j}$ from P_{j+1}, and then sets $\mathsf{m}_z = \mathsf{m}_x\mathsf{m}_y + \sum_{k=1}^{3} \mathsf{m}_z'^{k}$. Hence, any offset/additive error in the received message immediately translates to the output m_z, and if P_j holds $\mathsf{m}_x + \delta$ where it should have m_x (symmetrically for m_y), this will cause it to compute

$$\mathsf{m}_z'^{j-1} - \delta\lambda_y^{j-1} \text{ instead of } \mathsf{m}_z'^{j-1},$$
$$\mathsf{m}_z'^{j+1} - \delta\lambda_y^{j+1} \text{ instead of } \mathsf{m}_z'^{j+1}, \text{ and}$$
$$\mathsf{m}_z + \delta\mathsf{m}_y - \delta\lambda_y^{j-1} - \delta\lambda_y^{j+1} \text{ instead of } \mathsf{m}_z.$$

In Tab. 5, we now showcase any offsets the individual parties have on their shares throughout the circuit evaluation.

Again, P_1 includes offset $+1$ in its message which causes P_3 to have the same offset on m_c. This is simply carried through the addition gate $e = c + d$. Hence, the error propagates into multiplication $i = ef$ as it did in the prior example. In multiplication $g = ce$, there is an error in the m_v-parts of P_3 on both inputs, so that the output errors for both cases are aggregated. For both multiplications, the error propagates from the $\mathsf{m}_v'^2$ parts that P_3 sends to P_2 into the m_v-parts of P_2. For the final multiplication $j = hi$, P_2 has an offset of $\lambda_f^2\lambda_h^1$ in its message to P_1 (the first non-hash message deviating from an honest execution that P_1 receives). P_3 instead has an offset of $-(\mathsf{m}_f - \lambda_f^1 - \lambda_f^2)\lambda_h^1$ in the respective value that it includes in hash h_1. We leave out the other offsets here for space reasons, as they do not influence any messages sent towards P_1.

Table 5. Offsets/additive errors on the locally held and used values for all parties for evaluating the circuit from Fig. 13. "-" denotes no offset and "..." denotes an offset omitted due to being of no relevance for the attack and space reasons.

Wire	P_1			P_2			P_3		
	m'^2_v	m'^3_v	m_v	m'^3_v	m'^1_v	m_v	m'^1_v	m'^2_v	m_v
$c = ab$	-	$+\mathbf{1}$	-	-	-	-	-	-	$+1$
$e = c+d$			-			-			$+1$
$i = ef$	-	-	-	-	-	$-\lambda^2_f$	$-\lambda^1_f$	$-\lambda^2_f$	$+\mathsf{m}_f-\lambda^1_f-\lambda^2_f$
$g = ce$	-	-	-	-	-	$-\lambda^2_c-\lambda^2_e$	$-\lambda^1_c-\lambda^1_e$	$-\lambda^2_c$	$+\mathsf{m}_c-\lambda^1_c-\lambda^2_c$
								$-\lambda^2_e$	$+\mathsf{m}_e-\lambda^1_e-\lambda^2_e$
$j = hi$	-	-	$+\lambda^2_f\lambda^1_h$	...	$+\lambda^2_f\lambda^1_h$	...	$-(\mathsf{m}_f-\lambda^1_f-\lambda^2_f)\lambda^1_h$	...	...

For the final verification, P_1 computes for the values received from P_2

$$\mathsf{H}(\mathsf{m}'^1_c \| \mathsf{m}'^1_i \| \mathsf{m}'^1_g \| \mathsf{m}'^1_j + \lambda^2_f\lambda^1_h)$$

to compare to

$$h_1 = \mathsf{H}(\mathsf{m}'^1_c \| \mathsf{m}'^1_i - \lambda^1_f \| \mathsf{m}'^1_g - \lambda^1_c - \lambda^1_e \| \mathsf{m}'^1_j - (\mathsf{m}_f - \lambda^1_f - \lambda^2_f)\lambda^1_h)$$

received from P_3. P_1 could now test for $x, y, z \in \mathcal{R}$ if $\mathsf{H}(\mathsf{m}'^1_c \| \mathsf{m}'^1_i - x \| \mathsf{m}'^1_g - y \| \mathsf{m}'^1_j + \lambda^2_f\lambda^1_h - z) = h_1$ which holds (except for a hash collision) if and only if $x = \lambda^1_f, y = \lambda^1_c + \lambda^1_e, z = (\mathsf{m}_f - \lambda^1_f - \lambda^2_f)\lambda^1_h + \lambda^2_f\lambda^1_h$. From learning $x = \lambda^1_f$, P_1 can extract $f = \mathsf{m}_f - \lambda^1_f - \lambda^2_f - \lambda^3_f$. Furthermore, from learning $y = \lambda^1_c + \lambda^1_e$, it can extract $c + e = \mathsf{m}_c + \mathsf{m}_e + (\lambda^1_c + \lambda^1_e) + \lambda^2_c + \lambda^2_e + \lambda^3_c + \lambda^3_e$. Finally $z = (\mathsf{m}_f - \lambda^1_f - \lambda^2_f)\lambda^1_h + \lambda^2_f\lambda^1_h$ contains λ^1_h as only unknown since λ^1_f has already been extracted. Hence, P_1 learns that $\lambda^1_h(\mathsf{m}_f - \lambda^1_f) = z$ and, if $\mathsf{m}_f - \lambda^1_f \neq 0$, can extract, e.g., λ^1_h and then also h if $\mathcal{R}$ is a field. If $\mathcal{R} = \mathbb{Z}_{2^\ell}$ as originally used by Trident, it might still be possible to extract, e.g., the parity of h.

As the cheating now influences multiple (3) succeeding multiplications, the exhaustive search is over a domain of size $|\mathcal{R}|^3$ and, hence, more expensive. Still, for sufficiently small $\mathcal{R}$ and especially the binary domain $\mathcal{R} = \mathbb{Z}_2$, this can be done efficiently and with negligible risk of a hash collision.

C Details on Broken Robustness in Fantastic Four [15]

In this section, we give details beyond the high-level ideas and possible fixes from §4.2 on why the robustness mechanisms from Fantastic Four [15] are insufficient given Fantastic Four's delayed consistency checks.

Classic Robustness Recall from §4.2 that for a failed consistency check, Fantastic Four identifies the single, uninvolved party in this check as honest for its classic robustness approach. This party is then used as a trusted third party to finish the computation. We observe that this mechanism does not work when

delaying consistency checks as done in Fantastic Four. In Fig. 14, we provide an example for computing $c = a \cdot b$ and $e = c \cdot d$ (Fig. 2) where corrupt P_2 sends an incorrect value to P_1 for the first multiplication. Following the protocol specification (Fig. 7), this leads to P_1 sending two incorrect values to P_4 for the second multiplication.

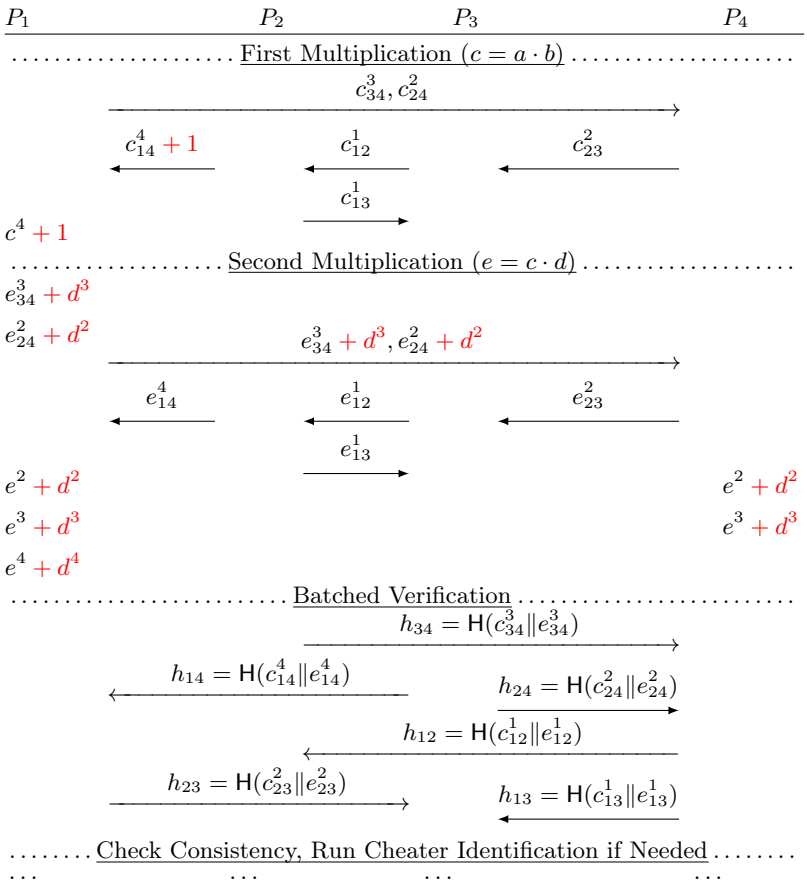


Fig. 14. Two multiplications and batched verification required to evaluate the circuit from Fig. 2 in Fantastic Four. Parts in red are errors introduced by cheating P_2 and we omit all locally held shares that do not contain any error.

We now have three hash inconsistencies in the verification:

$$\mathsf{H}(c^4_{14} + 1 \| e^4_{14}) \neq h_{14} = \mathsf{H}(c^4_{14} \| e^4_{14}), \quad \text{values from } P_2, P_3, \text{ detected by } P_1$$
$$\mathsf{H}(c^3_{34} \| e^3_{34} + d^3) \neq h_{34} = \mathsf{H}(c^3_{34} \| e^3_{34}), \quad \text{values from } P_1, P_2, \text{ detected by } P_4$$
$$\mathsf{H}(c^2_{24} \| e^2_{24} + d^2) \neq h_{24} = \mathsf{H}(c^2_{24} \| e^2_{24}), \quad \text{values from } P_1, P_3, \text{ detected by } P_4$$

The first inconsistency is between messages that P_1 receives from P_2, P_3, leaving uninvolved P_4. The other inconsistencies are between messages that P_4 receives from P_1 and P_2 respectively P_3, leaving uninvolved P_3 respectively P_2. Note that not only is there no unambiguous uninvolved party to be chosen as a trusted third party, but one of the options even is P_2 which is the cheater. Choosing P_2 would then disclose all private values to the only corrupted party P_2, yielding the worst possible breach of privacy.

Private Robustness The private robustness version of Fantastic Four aims to identify a set of at most two parties, containing the cheater, when one of the individual verifications fails (cf. §4.2). The procedure for that works as follows: Assume that P_i sends x to P_g while P_j during the verification provides a hash $h = \mathsf{H}(x)$ to P_g (details in [15]):

1. If $\mathsf{H}(x) \neq h$, P_g accuses P_i, P_j (via broadcast), providing $\mathbf{c}_i = \mathsf{H}(x)$ (x as received) and $\mathbf{c}_j = h$ (h as received).
2. If $\mathbf{c}_i = \mathbf{c}_j$, P_g has cheated in the prior step as there is no mismatch between $\mathbf{c}_i, \mathbf{c}_j$. Hence, P_g is identified as the cheater, and the cheater identification ends.
3. Otherwise, P_i accuses P_g if $\mathbf{c}_i$ does not match $\mathsf{H}(x)$ for x that it has sent to P_g. In this case, either P_g provided an incorrect $\mathbf{c}_i$ or P_i wrongfully accuses P_g. Hence, one of these two is the cheater.
4. If $\mathbf{c}_j$ does not match h that it has sent before to P_g, P_j accuses P_g. As in step 3, one of these two parties then is the cheater.
5. If P_i, P_j both accuse P_g, then clearly P_g is the cheater.
6. If neither P_i nor P_j accuses P_g, then the accusation raised by P_g either matches the values sent earlier by P_i, P_j (hence, they were inconsistent) or one of P_i, P_j cheats by not accusing P_g. In both cases, the cheater must be among P_i, P_j.

It is not elaborated in [15] how exactly cases with multiple failing verifications are handled. Yet, running the cheater identification on an arbitrary failed verification always yields a set containing the cheater, given that the honest parties have consistent states. This clearly is not the case if we delay and batch the verification. Revisiting the prior example of two multiplications and cheating P_2 given in Fig. 14, one inconsistency is that P_4 receives c_{24}^2 and incorrect $e_{24}^2 + d^2$ from P_1, while it receives $h_{24} = \mathsf{H}(c_{24}^2 \| e_{24}^2)$ from P_3. In the previously described cheater identification from [15], P_4 accuses P_1, P_3 with the inconsistent values they provided. Neither of these two parties accuses P_4, given that they receive (the hash of) what they have sent. Hence, $\{P_1, P_3\}$ is the identified set that should contain the cheater. Yet, P_2 is the cheater, which shows that the cheater identification from [15] is incorrect for delayed and batched verification.

D Details on Attacking and Fixing Quad [22]

Here, we provide the details of our attack on the fair four-party protocol Quad [22] as briefly outlined in §6. To keep consistent with the notation from [22], we call

the parties P_0, P_1, P_2, P_3 and the linear secret sharing semantics are as shown in Tab. 6. The main sharing semantic is $[\![v]\!]$ which, as one part, contains a sharing $\langle \lambda_v \rangle$ where P_0, P_3 additionally know both shares λ_v^1, λ_v^2 of.

Table 6. Secret sharing semantics for sharing a value $v \in \mathcal{R}$ in Quad [22].

Type	P_0	P_1	P_2	P_3	Correlation
$\langle v \rangle$	—	v^1	v^2	—	$v = v^1 + v^2$
$[\![v]\!]$	$(\mathsf{m}_v^*, \lambda_v^1, \lambda_v^2)$	$(\mathsf{m}_v, \lambda_v^*, \lambda_v^1)$	$(\mathsf{m}_v, \lambda_v^*, \lambda_v^2)$	$(\lambda_v^*, \lambda_v^1, \lambda_v^2)$	$v = \mathsf{m}_v - \lambda_v^1 - \lambda_v^2 = \mathsf{m}_v^* - \lambda_v^*$

The intuition to multiply $[\![x]\!], [\![y]\!]$ into $[\![z]\!]$ is similar to that of [13] in §3:

$$\begin{aligned}\langle m_z \rangle &= \langle x \cdot y + \lambda_z \rangle = \langle (\mathsf{m}_x - \lambda_x) \cdot (\mathsf{m}_y - \lambda_y) + \lambda_z \rangle \\ &= \mathsf{m}_x \mathsf{m}_y - \mathsf{m}_x \langle \lambda_y \rangle - \mathsf{m}_y \langle \lambda_x \rangle + \langle \lambda_x \lambda_y + \lambda_z \rangle.\end{aligned}$$

The components of mask λ_z are randomly sampled non-interactively so that P_0, P_3 learn all shares. The component $\langle \lambda_x \lambda_y + \lambda_z \rangle$ is substituted by $\langle \gamma_{xyz} \rangle$ that can be set up by P_0, P_3, each of them knowing all involved shares.[9] Regarding the detection of a corrupt party cheating, we focus here specifically on cheating from P_1, P_2 as they are active throughout the online phase. Both locally compute $\overline{\mathsf{m}}_z := \mathsf{m}_z + \lambda_z^*$, where P_1, P_2, P_3 sample λ_z^* non-interactively. In addition, the preprocessing is done so that P_0 is also able to compute $\overline{\mathsf{m}}_z$. The three parties then exchange their hashes of $\overline{\mathsf{m}}_z$ to check for pairwise consistency. The protocol description is in Fig. 15 and further details, including how exactly the consistency check on $\overline{\mathsf{m}}_z$ ensures that cheating is detected, can be found in [22].

Protocol $\Pi_{\mathsf{mult}}([\![x]\!], [\![y]\!]) \to [\![z]\!]$ of Quad [22]

Input: $[\![\cdot]\!]$-shares of $x, y \in \mathcal{R}$.

Output: $[\![\cdot]\!]$-shares of $z = x \cdot y$.

Preprocessing:

1. Non-interactively sample $\lambda_z^1, \lambda_z^2, \lambda_z^* \leftarrow\!\!\$\ \mathcal{R}$ so that they are distributed as per Tab. 6.

3. Run further operations, providing the following preprocessing material: (details in [22])
   - P_1, P_2, P_3 have $r_{1,2,3} \leftarrow\!\!\$\ \mathcal{R}$.
   - P_0, P_3 have $M_{\{3\}} = \lambda_x(\lambda_y - \lambda_y^*) - \lambda_y \lambda_x^* - \lambda_z^* + r_{1,2,3}$.
   - $\langle \gamma_{xyz} \rangle$ have $\psi \in \mathcal{R}$, where P_0, P_3 also have all shares.

Online:

[9] Note that [22] does not use the notation γ_{xyz}, but instead lets P_0, P_3 non-interactively sample $r_{0,1,3}$ together with P_1 and then send $M_{\{0,3\}} = \lambda_z + \lambda_x \lambda_y + r_{0,1,3}$ to P_2. We use the less verbose notation $\langle \gamma_{xyz} \rangle$, interpreting $\gamma_{xyz}^1 = -r_{0,1,3}$ and $\gamma_{xyz}^2 = M_{\{0,3\}}$, renaming some of the variables originally used in [22] to enhance readability.

1. P_1, P_2 compute $V_{\{1,2\}} = \mathsf{m}_x \mathsf{m}_y$ and $M_{\{1,2\}} = V_{\{1,2\}} + r_{1,2,3}$.
2. P_i for $i \in \{1,2\}$ computes $M_{\{i\}} = \mathsf{m}_x \lambda_y^i + \mathsf{m}_y \lambda_x^i - \gamma_{xyz}^i$ and $V_{\{i\}} = V_{\{1,2\}} - M_{\{i\}}$.
3. P_1, P_2 exchange $M_{\{1\}}, M_{\{2\}}$.
4. P_i for $i \in \{1,2\}$ computes $\mathsf{m}_z = V_{\{i\}} - M_{\{3-i\}}$.

Verify:

1. P_2 sends $M_{\{1,2\}}$ to P_0.
2. P_0 computes $\mathsf{m}_z^* = M_{\{1,2\}} - (\mathsf{m}_x^* \lambda_y + \mathsf{m}_y^* \lambda_x + M_{\{3\}})$.
3. P_0 computes $\overline{\mathsf{m}}_z = \mathsf{m}_z^* + \lambda_z$ and P_1, P_2 compute $\overline{\mathsf{m}}_z = \mathsf{m}_z + \lambda_z^*$.
4. The following sets of parties compute the following messages and exchange them with anyone in their set. They abort in case of a mismatch:
   - $\{P_0, P_1\}$: $h_{01} = \mathsf{H}(M_{\{1,2\}})$.
   - $\{P_2, P_3\}$: $h_{23} = \mathsf{H}(\gamma_{xyz}^2)$.
   - $\{P_0, P_1, P_2\}$: $h_{012} = \mathsf{H}(\overline{\mathsf{m}}_z)$.

Fig. 15. Multiplication protocol of Quad [22] (some operations inlined and restructured according to the optimizations that [22] describes in text).

The verification again is delayed until the end of the protocol, right before the output phase, batching consistency checks over all multiplications. As for [13] in §3 and for [15] in §4, we compute $c = a \cdot b$ and $e = c \cdot d$ (Fig. 2). This time, we let a corrupt P_1 inject an offset of $+1$ when sending $M_{\{1\}}$ to P_2 during the first multiplication. Resulting deviations from the protocol execution (excluding preprocessing) are depicted in Fig. 16.

At the end of the first multiplication, P_2 holds $\mathsf{m}_c - 1$ instead of m_c. This results in many inconsistencies during the second multiplication and verification as depicted in Fig. 16. P_2 will receive h_{012} from P_0 which takes as one input correct $\overline{\mathsf{m}}_c$, inconsistent to $\overline{\mathsf{m}}_c - 1$ that P_2 holds. Hence, the verification will still detect the cheating. Yet, when P_2 computes $M'_{\{2\}}$ for the second multiplication, this has a summand $\mathsf{m}_c \lambda_d^2$ and the offset of -1 on m_c will result in an offset of $-\lambda_d^2$ on the message sent to P_1. Furthermore, as $M'_{\{1,2\}}$ of the second multiplication contains summand $\mathsf{m}_c \mathsf{m}_d$, P_2 instead computes $M'_{\{1,2\}} - \mathsf{m}_d$ due to the offset on m_c. This is sent to P_0, which computes $\overline{\mathsf{m}}_e - \mathsf{m}_d$ instead of $\overline{\mathsf{m}}_e$ because the offset directly translates. P_0 then sends $h_{012} = \mathsf{H}(\overline{\mathsf{m}}_c \| \overline{\mathsf{m}}_e - \mathsf{m}_d)$ to P_1.

Now, P_1 has $h_{012} = \mathsf{H}(\overline{\mathsf{m}}_c \| \overline{\mathsf{m}}_e - \mathsf{m}_d)$, $\overline{\mathsf{m}}_c$, m_d, λ_d^1, and $\overline{\mathsf{m}}_e + \lambda_d^2$ instead of $\overline{\mathsf{m}}_e$, caused by P_2 sending $M'_{\{2\}} - \lambda_d^2$ instead of $M'_{\{2\}}$ before. To test for an arbitrary value $x \in \mathcal{R}$ for d, it can test if

$$\begin{aligned} h_{012} &= \mathsf{H}(\overline{\mathsf{m}}_c \| (\overline{\mathsf{m}}_e + \lambda_d^2) - 2\mathsf{m}_d + \lambda_d^1 + x) \\ \implies \overline{\mathsf{m}}_e - \mathsf{m}_d &= \overline{\mathsf{m}}_e + \lambda_d^2 - 2\mathsf{m}_d + \lambda_d^1 + x \\ \implies 0 &= \lambda_d^2 - \mathsf{m}_d + \lambda_d^1 + x \\ \implies \mathsf{m}_d &= x + \lambda_d^1 + \lambda_d^2 \implies d = x, \end{aligned}$$

extracting d and breaking the security of the protocol.

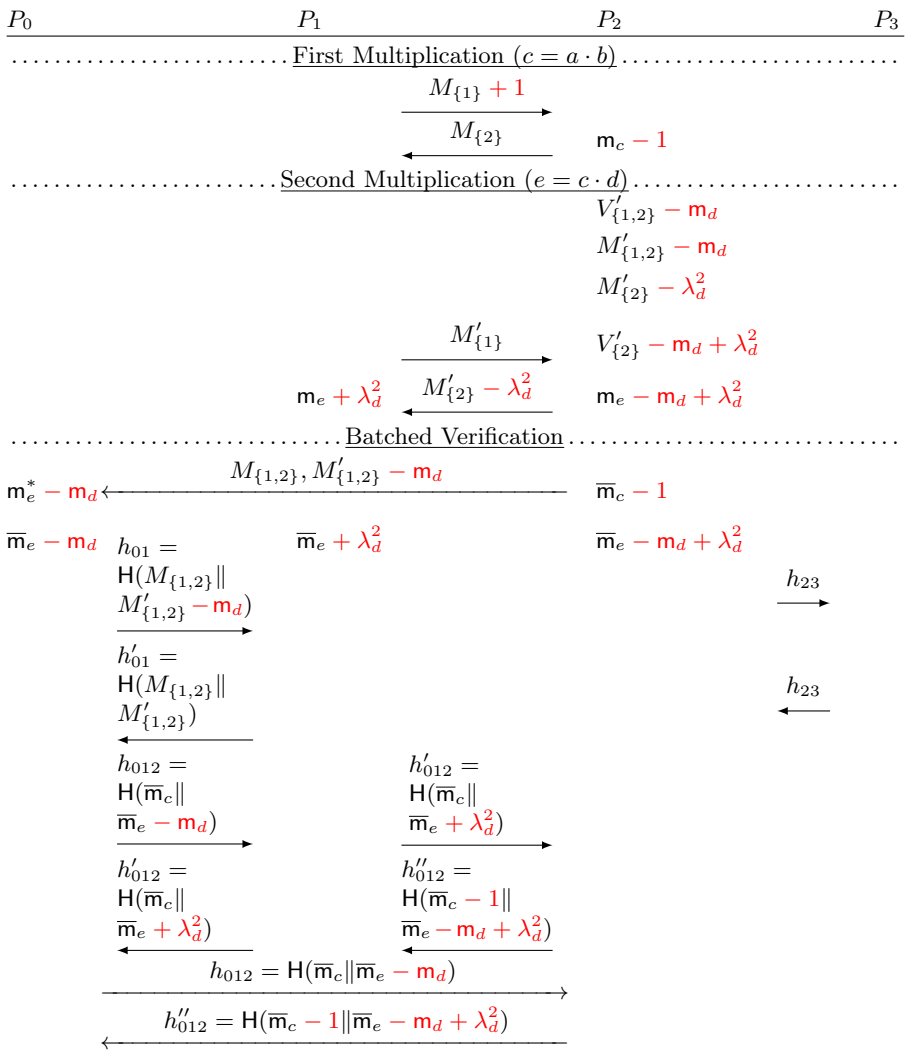


Fig. 16. Online and verification phases of two multiplications and batched verification required to evaluate the circuit from Fig. 2 in Quad [22]. Parts in red are errors introduced by cheating P_1 and we omit all locally held shares that do not contain any error. We also omit the comparison of hash values inside the batched verification as it plays no role in our attack. Intermediate values which do not have a unique notation, e.g., $M_{\{1\}}$, are labeled, e.g., $M'_{\{1\}}$ for the second multiplication.

D.1 Fixing Quad

Again, the high-level idea for fixing Quad is to use $\mathcal{F}_{\text{CheckEqs}}^{A,(s_a)_{a\in A},(t_a)_{a\in A}}$ from §3.5 that aggregates consistency checks. This needs to check equality of h_{01} as pro-

vided by P_0, P_1 and h_{23} as provided by P_2, P_3 which is easily integrated in a call to $\mathcal{F}_{\text{CheckEqs}}^{A,(s_a)_{a\in A},(t_a)_{a\in A}}$. Furthermore, checking equality between h_{012} as provided by P_0, P_1, P_2 can be split into three pair-wise checks. We call the resulting specific functionality $\mathcal{F}_{\text{CheckEqs}}^{\text{Quad}}$. If two honest parties hold inconsistent h_{012}, the corresponding pair-wise check will fail. Furthermore, per the proof of [22], any cheating attempt before the consistency check causes at least one pair of honest parties to use inconsistent hashes in the consistency check, leading to detection. This provides the core argument why Quad can be successfully fixed using the modified verification in Fig. 17. The formal proof would be close to the proof in §3.5, so we omit the details here.

Protocol Verification step of $\Pi_{\mathsf{mult}}^{\text{FIXED}}(\llbracket x \rrbracket, \llbracket y \rrbracket) \to \llbracket z \rrbracket$ for Quad [22]

Verify (batched):

1. P_2 sends $M_{\{1,2\}}$ to P_0.
2. P_0 computes $\mathsf{m}_z^* = M_{\{1,2\}} - (\mathsf{m}_x^*\lambda_y + \mathsf{m}_y^*\lambda_x + M_{\{3\}})$.
3. P_0 computes $\overline{\mathsf{m}}_z = \mathsf{m}_z^* + \lambda_z$ and P_1, P_2 compute $\overline{\mathsf{m}}_z = \mathsf{m}_z + \lambda_z^*$.
4. Provide inputs to $\mathcal{F}_{\text{CheckEqs}}^{\text{Quad}}$:
   - P_0 provides $h_{01}^1 = \mathsf{H}(\dots \| M_{\{1,2\}} \| \dots)$, P_1 provides $h_{01}^2 = \mathsf{H}(\dots \| M_{\{1,2\}} \| \dots)$.
   - P_2 provides $h_{23}^1 = \mathsf{H}(\dots \| \gamma_{xyz}^2 \| \dots)$ and P_3 provides $h_{23}^2 = \mathsf{H}(\dots \| \gamma_{xyz}^2 \| \dots)$.
   - P_0 provides $h_{012a}^1 = \mathsf{H}(\dots \| \overline{\mathsf{m}}_z \| \dots)$ and P_1 provides $h_{012a}^2 = \mathsf{H}(\dots \| \overline{\mathsf{m}}_z \| \dots)$.
   - P_0 provides $h_{012b}^1 = \mathsf{H}(\dots \| \overline{\mathsf{m}}_z \| \dots)$ and P_2 provides $h_{012b}^2 = \mathsf{H}(\dots \| \overline{\mathsf{m}}_z \| \dots)$.
   - P_1 provides $h_{012c}^1 = \mathsf{H}(\dots \| \overline{\mathsf{m}}_z \| \dots)$ and P_2 provides $h_{012c}^2 = \mathsf{H}(\dots \| \overline{\mathsf{m}}_z \| \dots)$.
5. P_i proceeds if it receives `Accept` from $\mathcal{F}_{\text{CheckEqs}}^{\text{Quad}}$ and aborts otherwise.

Fig. 17. Fixed, batched verification of the multiplication of Quad [22] in Fig. 15. Labels $012a, 012b, 012c$ for the exchanged hashes represent the three pair-wise comparisons that we have split the original check on h_{012} into.

To instantiate $\mathcal{F}_{\text{CheckEqs}}^{\text{Quad}}$, we can use a variant of Quad where the original verification is not delayed. This is secure as per the proofs in [22] which only neglect to include the delays.

E Comparison to the Double-Dipping Attack [20,16]

The protocol of [20] aims to enable unconditionally secure and robust MPC in an honest super-majority (i.e., an adversary can only corrupt less than a third of all parties) where the total communication per multiplication scales linearly with the number of parties n, even for deep and narrow circuits. While batching can enable a secure opening required for DN-style [17] multiplications with only linear communication, sufficient batching is not possible for very few multiplication gates at the same multiplicative depth. On a high level, the approach of [20] is to *optimistically* evaluate segments of the circuit first with linear communication. After each segment, all openings of that segment are repeated, now in an actively secure and batched fashion, to check if the prior optimistic openings

were done correctly. Using sufficiently large segments enables sufficient batching in the actively secure opening step, yielding linear communication.

Naïvely executing the optimistic phase would enable the *double-dipping* attack. A corrupt party P_{king} tasked with reconstruction of a value (for a multiplication) may not correctly distribute it to all other parties, but hand an incorrect result to some honest parties. These honest parties eventually compute an incorrect output share of the multiplication while all other parties hold the correct shares. For another multiplication in the same segment which uses the output of the prior multiplication, the honest parties will send inconsistent values to P_{king}. Yet, due to redundancy in the honest super-majority setting, P_{king} is able to compute the correct values of these honest parties too. The difference between both versions now enable the adversary to reconstruct the value on the other input wire to the second multiplication.

There are notable similarities between the double-dipping attack in [20] and our attack. For both attacks, one or a few honest parties are sent incorrect data to create some inconsistency. Due to optimistic execution, this is not detected immediately and influences further operations, where the parties that have received incorrect data will deviate from the protocol. Redundancy then enables the adversary to extract knowledge by comparing the behavior of deviating parties to what they would be supposed to send.

Yet, there also are significant differences. Centrally, our attack targets redundancy provided by the verification, whereas the double-dipping attack in [20] already works on the optimistic execution alone. Generally, the differences stem from significantly different settings and goals. The goal of [20] is good scalability for many parties. Hence, they aim to achieve asymptotic communication that is linear in the number of parties. In contrast, the works considered here [13,15,29,22] aim for high concrete performance with a low and fixed number of parties. A honest super-majority is required by [20], whereas this is not generally the case in our work.[10] The works affected by our attack all add redundancy by each message to be sent being computed by two parties, one to send the message and one to later use it in the batched and hash-based consistency check with zero amortized overhead. In contrast to that, [20] adds redundancy by utilizing error-correcting codes. This redundancy and the resulting double-dipping attack are in the optimistic execution, whereas our attack uses redundancy which is only introduced later in the consistency check. The double-dipping attack from [20] also would be easier to carry out than our attack. This is as redundant messages in our attacks are compressed into a single hash, whereas they are available in the clear in [20]. Causing an inconsistency at an early multiplication can cause inconsistencies for many future multiplications. This makes it harder to extract infor-

[10] SWIFT [29] considers one corruption among three parties. While the protocol (except for its inputs phase) appears to evade the attack (cf. §5.1), this is due to an independent optimization and not due to the setting. E.g., Trident [13] is fully vulnerable in the online phase and uses three parties and a dealer. One could simply distribute the dealer, or leave out the independent optimization in SWIFT, yielding vulnerable three-party protocols.

mation as all redundant material is compressed to one hash (cf. §B), or it requires to target later multiplications. In [20], the redundant material is not compressed.

The issue is fixed in [20] by removing redundancy in the optimistic execution. This is done by switching to a full threshold sharing and later using the original threshold for the verification. While the protocols considered in our work do not use full threshold sharings for the optimistic execution, redundancy in the communication is only introduced during verification. Hence, there is no use of applying the fix from [20], and it would also decrease performance as reconstructing values from full threshold sharings requires more communication. Furthermore, the protocol from [20] essentially opens each value twice, once during optimistic execution and once during verification. While this makes sense to achieve good asymptotic performance for many parties, the protocols considered in our work aim for good concrete performance for few parties, avoiding the necessity to open the same values twice. Our fix maintains that the protocols open values only once for each multiplication and has zero amortized overhead for the verification. On the other side, we do not identify any direct benefit of using a variant of our fix to improve [20]. This is as our fix aims for good concrete performance for few parties and only a single corruption, whereas [20] has significantly different requirements.

Another possible solution for the double-dipping attack has been presented in [16]. In contrast to [20], it removes redundancy from the optimistic execution by only using a subset of parties there. The verification relies on zk-FLIOPs [5]. For concrete efficiency, it is preferable to avoid the overhead from using zk-FLIOPs in the online phase where possible, which is also done in SWIFT [29].