To Catch a Cheater

Some notes on the malicious (in)security of ORQ

Eli Baum

Boston University

Abstract. Malicious-secure multiparty computation protocols protect against an adversary's arbitrary misbehavior. In the honest-majority four-party setting, one common technique relies on all communication between parties being duplicated. Under this approach, all correct messages are sent twice, while corrupted messages are only sent by the adversary and will not match concurrent correct messages. When a receiver observes that inconsistency, it announces cheating has occurred (and possibly aborts). Existing implementations often optimize this procedure by batching many such checks into a single hash and running a final consistency check just before revealing the result of a computation.

Brüggemann and Schneider (Eurocrypt 2026) recently showed that these delayed consistency checks in honest-majority, malicious-secure protocols can violate privacy. Adversaries can introduce errors such that subsequent *incorrect* hashes reveal their missing secret share and allow plaintext secrets to be recovered just before the honest parties abort. Their suggested fix evaluates the hash comparison under multiparty computation, rather than in plaintext. In this report, we detail our fix for the Fantastic Four protocol in Orq, a recent system for secure analytics that is vulnerable to the attack. The new implementation has a modest overhead that amortizes with larger inputs. The complexity of the modified protocol highlights the difficulty of implementing malicious-secure systems in practice; even seemingly harmless optimizations can break privacy.

1 Introduction

At Eurocrypt 2026, Brüggemann and Schneider [BS26] presented an ingenious attack on the delayed consistency checks of many honest-majority, malicious-secure multiparty computation protocols. These protocols rely on duplicated messages to constrain an adversary's behavior. With four computing parties, at most one can be corrupted by the adversary under the honest-majority assumption; thus, if all messages are sent by at least two parties, any message deviating from the protocol will necessarily be caught. However, as first observed in [BS26], even when the malicious party is caught, the check itself may leak sensitive information before the honest parties abort.

In this report, we describe how we implemented the fix proposed in that paper within Orq [BBM+25], our recent framework for oblivious relational analytics under MPC. Here, we focus on the "Fantastic Four" four-party protocol by Dalskov et al. [DEK21], one of those which suggests using delayed consistency checks. In Orq, we had implemented two versions of Fantastic Four: one with the delayed consistency checks, and one where the receiver confirms that replicated messages match immediately upon receipt. Consistent with the observations of [BS26], only the former implementation appears vulnerable to their attack. Thus, we focus this report on that protocol.

E-mail: elibaum@bu.edu (Eli Baum)

In the following sections, we provide a brief overview of the Fantastic Four protocol and its recent attack (§3), discuss the details of our fix (§4), and conclude with a performance evaluation on sorting and query benchmarks (§5). We also propose new functionalities, and protocols instantiating them (Appendix A), to begin formalizing the deferred-check protocol.

2 Notation

For integers $a < b$, $[a, b]$ refers to the set $\{a, a+1, \ldots, b-1, b\}$. All operations occur over finite rings $\mathbb{Z}_{2^k}$ or fields $\mathbb{F}_{2^k}$. In our context, a *secret sharing of* x consists of four shares $[x]_0, [x]_1, [x]_2, [x]_3$ such that $\sum_i [x]_i = x$, and party P_i holds shares $\{[x]_j : j \neq i\}$ (that is, $[x]_i$ is the share *not* known to P_i). To refer to the secret sharing, in general (as opposed to any particular share), we write $[x]$. $[x]$ may be *arithmetic* (in which case the reconstruction operator is $+$ over some ring, usually $\mathbb{Z}_{2^k}$) or *boolean* (with reconstruction operator bitwise $\oplus$). We can also view boolean sharing over $\mathbb{F}_{2^k}$, where $+ \equiv \oplus$.

The Fantastic Four protocol defines a subroutine `JMP` (joint message passing) between three parties (P_s, P_r, P_h): for some x known to (P_s, P_h), P_s sends x to P_r (the receiver), while P_h stores the hash $H(x)$. P_r also computes and stores the hash. We will compactly represent this party assignment as $[s \to r; h]$.

3 Fantastic Four and Attacks on Delayed Consistency Checks

Malicious security in multiparty computation often comes at a price, but this need not always be a practical barrier to implementations. Dalskov et al. [DEK21] introduced the "Fantastic Four" protocol for malicious security with an honest majority (that is, only a single party may be corrupted). It is nearly as efficient as an equivalent semi-honest protocol, incurring only slightly greater communication.

Dalskov et al. also propose an extension of the protocol that provides *robustness*, or guaranteed output delivery. [BS26] also identify an attack on the robustness protocol that allows the malicious party to *frame* an honest party. However, since Orq only provides security up to abort – rather than guaranteed output delivery – we do not consider that attack here.

Multiple versions of Fantastic Four exist in the open-source literature. We are aware of implementations in MP-SPDZ [Kel20], HPMPC [HKSW+25], and Orq [BBM+25]. Here, we focus on Orq, which has not yet fixed the attack of [BS26]. Neither MP-SPDZ nor HPMPC have implemented the constant-round fix described by [BS26], so they may incur greater overheads than necessary.

Underspecified Protocol Primitives. The original Fantastic Four paper does not fully specify all necessary primitives. For example, while the secret sharing *scheme* is described, the secret-sharing protocol itself is not. Straightforward implementations in the outsourced setting are possible, but to ensure both operations are fully general, we must also ensure *consistency* of the resulting secret shares, something Orq's existing implementation of `secret_share` does not do. As part of our fix, we define a *consistent secret sharing* functionality, $\mathcal{F}_{\mathsf{ConsistentShare}}$ (Figure 2), and implement a protocol where receiving parties ensure the sending party provided matching shares (Figure 10).

While Orq uses `secret_share` throughout the internals of the framework (e.g., within sorting and shuffling), we observe that all current uses can be replaced by a non-interactive version that is malicious-secure by default. Orq terms this primitive `public_share`, and assumes a plaintext that is known by a qualified subset of the parties. We replace all

of Orq's internal calls to `secret_share` with `public_share` to avoid having to incur the cost of consistent secret sharing. $\mathcal{F}_{\mathsf{ConsistentShare}}$ is now only invoked within our fixed protocol. However, any future protocols with internal calls to `secret_share` must also use $\mathcal{F}_{\mathsf{ConsistentShare}}$ if `public_share` is not applicable.

We have also redesigned the malicious-protocol API of Orq. Previously, Orq provided a single `malicious_check` function that users were expected to manually call just before the end of a computation. We add a new Verifier protocol, which provides `start_malicious_check` and `finalize_malicious_check` methods, and does not use any batching optimizations internally (avoiding the attack in question). In addition to rectifying these issues, we also:

- ensure calls to `open` run malicious checks before revealing plaintext, and that the resulting shares are consistent.
- add a new malicious-secure commitment functionality, which is used to configure all shared PRG seeds.
- implement malicious-secure binary-to-arithmetic conversion (`b2a`).
- run checks *within* sorting algorithms to guarantee malicious security (see Appendix B)
- improve code clarity by factoring out common functionality in various multiplication-style operations (dot product, ring and field multiplication, bitwise AND).

We hope that these changes will streamline the implementation of other malicious protocols that might be added to the Orq framework in the future.

The Attack. In the interest of brevity, we omit a full discussion of the multiplication protocol of Fantastic Four; for more details, see that paper [DEK21]. For our purposes, it suffices to recognize that each cross-product $[x]_i[y]_j + [x]_j[y]_i$ is known to *two* parties (namely, distinct parties P_g, P_h such that $\{g,h\} \cap \{i,j\} = \varnothing$). Honest P_g and P_h both send the *same* message to P_i (and respectively, P_j), allowing the receiver to catch a deviating message. This share-replication property holds for all messages exchanged in the protocol.

A simple optimization has only one party send each message m in full, while the other sends a hash $H(m)$, for some collision-resistant hash function $H(\cdot)$. Heavily vectorized computations, like those presented in Orq, allow the resulting communication overhead to be amortized.

Fantastic Four and other recent malicious-secure protocols propose *batching* such hash checks: a single hash can be aggregated and checked once before outputs are revealed. To the extent that prior work provides rigorous security analyses, they fail to include this batching operation, viewing it as a mere implementation detail. Unfortunately, [BS26] demonstrated that deferred batch checks are *not* secure, in general: while these checks will always *catch* malicious behavior, they may not do so while protecting privacy.

The key insight of [BS26] is that the malicious party can introduce an additive error into an *honest* party's share such that future messages from that honest party leak information about the adversary's missing share. With some knowledge of a domain restriction on the underlying secret (for example, "$x \in \{0,1\}$"), the adversary can easily run an exhaustive search over all possible hash values. Even over larger domains, efficient adversaries can still *rule out* possible plaintexts, clearly breaking privacy.

Without loss of generality, assume P_0 is malicious.[1] It does not know $[x]_0$ for any secret shared value x. [BS26] observe that P_0 can introduce an additive error of $+1$ in one

[1]The following considers a particular topology of sending, hashing, and receiving parties. The attack holds with full generality, but perhaps just with different share or party indices.

Functionality $\mathcal{F}^{A}_{\mathsf{CheckEqs}}$

INPUT: d_a^H from P_h and d_a^R from P_r, for all assignments $a = [s \to r; h] \in A$ (where P_s sends to the receiver P_r, and P_h hashes)

OUTPUT: accept if all hashes match ($\forall a \in A : d_a^H = d_a^R$); otherwise, reject.

Figure 1: A restatement of the equality-check functionality from [BS26, Figure 4]

Functionality $\mathcal{F}_{\mathsf{ConsistentShare}}$

PARAMETER: Secret-sharing ring $\mathcal{R}$

INPUT: message m from party P_i

OUTPUT: all parties receive a uniformly random replicated secret sharing $[m]$ over $\mathcal{R}$ subject to $[m]_i = 0$.

Figure 2: Consistent Sharing

of the honest parties' shares (say, P_1) while computing a multiplication gate $[z] = [x \cdot y]$. Then, P_1 nominally holds shares of $(z + 1)$, while all other parties hold shares of z. If this value is used as input to another multiplication gate (say, $[q] = [z \cdot w]$, where w is known to be a bit), P_1 will send P_0 a hash of its (incorrect) share of $[(z + 1) \cdot w] \equiv [zw] + [w]_0$, while other parties will send P_0 shares of $[zw]$. Now, P_0 holds shares (w_1, w_2, w_3) and knows $\sum_i w_i \in \{0, 1\}$. Thus, $w_0 \in \{0, 1\} - (w_1 + w_2 + w_3)$, and the adversary need only compute two possible hashes and compare them to that which it received from P_1. See [BS26, Figure 8] for a detailed illustration.

The solution, while conceptually straightforward, requires some care to implement correctly. In short, we run the hash comparison *under MPC*, thus never leaking a possibly-diverging hash value to the malicious party. The only plaintext output is the single bit of information denoting whether *all* hashes matched, as opposed to the hash values themselves, or even individual hash-check results. [BS26] have implemented a proof-of-concept of their attack on MP-SPDZ's implementation of Fantastic Four and demonstrated their ability to recover plaintext. Next, we provide details of our fix and how we implemented it in ORQ's codebase.

4 Fixing Delayed Verification

In this section, we walk through the malicious-secure fix for delayed verification. We restate $\mathcal{F}_{\mathsf{CheckEqs}}$ of [BS26, Figure 4] in Figure 1. While that paper follows [DEK21] and uses a six-hash version of the check, ORQ sets A to contain all twelve pairs of parties, as is required for honest-majority malicious shuffle.

The general problem we need to solve is how to compare two bit strings held by different parties, while handling various failure modes of inconsistent inputs and maliciously-seeded PRGs. At a high level, we will (1) reshare each hash to the other parties and XOR pairs together, (2) securely generate a random secret-shared vector, and (3) compute a random linear combination. If all malicious checks pass, then the two hashes in each pair will be the same. Then, their XOR will be zero, as will be the linear combination, with negligible soundness error $1/|\mathbb{F}|$ (assuming a collision-resistant hash function).

The constant-round check proposed by [BS26] has each pair of parties knowing d_a

Functionality $\mathcal{F}_{\mathsf{Random}}$

PARAMETER: ring $\mathcal{R}$, seed space $\mathbb{S}$, pseudorandom generator family $G_m : \mathbb{S} \to \mathcal{R}^m$

INPUT: $n \in \mathbb{Z}$, length of vector to generate.

Without loss of generality, assume the adversary controls P_i.

- Choose seeds $s_j, s_g, s_h \xleftarrow{\$} \mathbb{S}$ where $\{i, j, g, h\} = \{0, 1, 2, 3\}$
- Compute $[\vec{r}]_x \leftarrow G_n(s_x)$ for each $x \in \{j, g, h\}$
- Sample a random vector $[\vec{r}]_i \xleftarrow{\$} \mathcal{R}^n$.

OUTPUT: the secret sharing of $[\vec{r}] \xleftarrow{\$} \mathcal{R}^n$ to each party, and send (s_j, s_g, s_h) to the adversary

Figure 3: Four-party random-share generation functionality

secret share their hashes[2] and XOR them together. Let $\vec{d}$ be a vector of these hashes in some canonical ordering. Two parties – the receiver P_r and hasher P_h – hold hash d_a, for $a := [s \to r; h]$. We refer to these two digests as d_a^R and d_a^H, respectively, and generally refer to the entire vector of digests held by all receivers, $\vec{d}^R$, and all hashers, $\vec{d}^H$. If no malicious behavior occurred, we have $\vec{d}^R = \vec{d}^H$. To check this equality with malicious security, parties sample a random secret-shared vector $[\vec{r}] \in \mathbb{F}$ and compute the dot product $z := \langle \vec{r}, \vec{d}^R \oplus \vec{d}^H \rangle$. z will be zero if the check passes and uniformly random over $\mathbb{F}$ otherwise (thus, non-zero with all but negligible probability).

Why must we sample *secret-shared* random values? This prevents the malicious party from learning a corrupted hash in the clear: if only a single hash value d_a is incorrect, then the (non-zero) output is $z = r_a \cdot (d_a^R \oplus d_a^H)$. The malicious party can compute $z \cdot r_a^{-1}$ and learn the *difference* between the expected and received hashes. This would allow a variant of the original attack of [BS26] to go through.

4.1 Sampling Random Values

The functionality $\mathcal{F}_{\mathsf{Random}}$ for sampling secret-shared random vectors is given in Figure 3 and instantiated following Π_{Random} in Figure 9. It is important that $\mathcal{F}_{\mathsf{Random}}$ provide PRG seeds for the adversary's shares, rather than generate all shares truly at random: otherwise, the adversary could distinguish the real and ideal worlds [AFL+16, §3.3], since no efficient simulator can invert a PRG.

We build a queue of commitments to shared PRG seeds. To generate a new PRG seed shared among a group g, each party opens their next commitment in the queue towards all other parties in the group. Parties check these openings against the previously-received commitments (presumed to be generated in preprocessing) and then XOR the individual seeds to form a joint seed. The values generated by this PRG are clearly pseudorandom, and, importantly, were entirely unpredictable prior to commitment opening. Thus, we enforce the requirement that any malicious behavior by the adversary could not have been adaptively chosen with knowledge of the eventual PRG output (as would have been an issue, if, say, a single, pre-shared PRG seed was used for all checks).

[2]The hash is also known to the sender. However, the hash check is meant to *catch* a malicious sender, so we do not consider the sender's view here.

If the resulting PRG values will never be revealed in plaintext, but remain in secret-shared form (as is the case in our protocol), then it is sufficient to open all PRG seed commitments in preprocessing: the adversary's three random shares give it no information about the underlying random value.

4.2 Round Complexity

[BS26] propose the linear combination check as a constant-overhead batch check. In this section, we show that a log-depth equality circuit, as implemented by HPMPC, could also provide competitive performance. Let k be the output bitwidth of the hash function. We note that this parameter is constant with respect to the operation of the underlying MPC protocol. Thus, asymptotically, both approaches have $O(1)$ rounds and amortized bandwidth overhead per check.

Constant-Overhead Check. Each of the steps of the random linear combination verification circuit require at least one round. We must (1) secret-share all hashes, (2) gossip received secret shares to other parties and confirm consistency, (3) compute their dot product with a random vector, and (4) open the result.

This gives four rounds, with overall bandwidth $O(|A| \cdot k)$. Our current implementation favors more interpretable code, so some steps are executed sequentially, and we open PRG seed commitments online rather than in preprocessing. As such, we currently require close to 30 network round trips. This may cause an unnecessary overhead in WAN, but even in the worst case, only $O(\log n)$ such checks are required for oblivious queries. Systems optimizations and API redesigns in the ORQ codebase could bring this value closer to its theoretical limit.

Log-Depth Equality Circuit. On the other hand, the theoretical minimum for a logarithmic-depth equality circuit is $\Omega(\log k)$ rounds and $\Omega(|A| \cdot k)$ bandwidth. Concretely, for the hash function used in this implementation (`blake2b`), $k = 256$, so we must incur at least 8 rounds to compare each element with zero.[3] A binary reduction tree over the twelve equality-check outputs costs another four rounds, and gives us a single bit which can be revealed to all parties. Overall, this approach requires at least 12 rounds. Unfortunately, due to architectural limitations in the ORQ framework, we were unable to implement this version of the check.

4.3 Details of our Approach

Our implementation of $\Pi^A_{\mathsf{CheckEqs}}$ is shown schematically in Figure 4 and described in Figure 5. We have designed this implementation to provide a minimally-invasive fix for the ORQ codebase. We do not modify the multiplication protocol of Fantastic Four, although we do generalize ORQ's implementation with a templated version, in order to take advantage of the repetition between the dot product, multiplication, and bitwise AND protocols. `JMP` remains as described in ORQ. We split ORQ's `malicious_check` primitive into two components, which simplifies the implementation of multithreaded checks. Each party creates an instance of a Verifier class, which handles the actual hash check.

All threads first call `start_malicious_check`. Verification flags are cleared; threads compute their overall hash digests $\vec{d}^H$ (hashes for which they were the hashing party) and $\vec{d}^R$ (hashes for which they were the receiving party) and forward them to their local Verifier instance. The sending party never computes a hash. Next, parties call

[3] ORQ's equality circuit has bandwidth $O(k \log k)$; the more complex Brent-Kung prefix network architecture [BK82] could achieve the minimum $O(k)$ bandwidth, but with twice as many rounds.

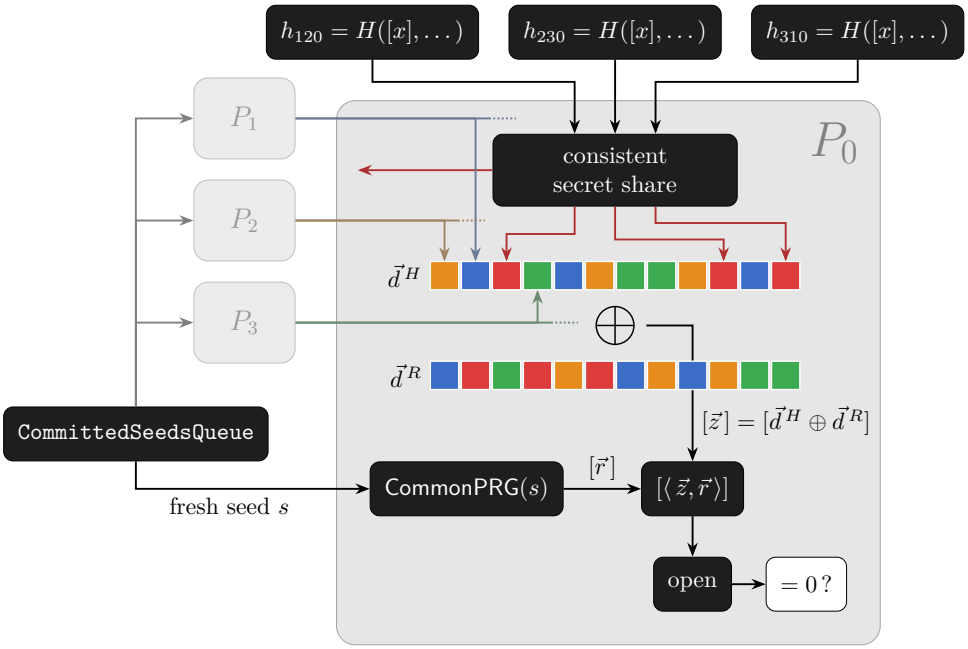


Figure 4: A schematic of our implemented fix. The result of the linear combination will be uniformly random if nonzero, so the zero check can be performed in plaintext. All parties evaluate the same protocol.

`finalize_malicious_check`, which executes Π_{CheckEqs}. Only the first thread executes over the aggregated hashes; all others wait for its return. Parties abort if reject is output.

As also observed by [KNOS25], concurrent malicious checks on shared state are not secure in general. While the single-threaded check implemented here may impose a small performance penalty, a multithreaded implementation could leak sensitive information if the adversary learned *which* thread crashed. Under Orq's multithreading model, for example, this could leak the location of a corrupted index within some vectorized computation.

5 Evaluation

We evaluate our fixed protocol on a variety of sorting workloads and five TPC-H queries from the Orq paper. Our implementation within the Orq framework provides full malicious security guarantees while incurring an acceptable performance overhead, although many further systems optimizations are possible. The new Verifier itself is about 250 lines of C++; the committed seeds queue, 280 lines. We will push all relevant code to the public Orq repository: https://github.com/CASP-Systems-BU/orq

All experiments are run on a cluster of four AMD EPYC 9655P nodes, each with 96 cores and 251 GiB of RAM, connected by a 25 Gbps, 0.25 ms LAN. We execute each experiment three times and report the minimum. Three versions of the Fantastic Four protocol are considered in our experiments: Baseline, which skips malicious checks (but still runs all other protocol components); our Fixed implementation; and Orq's `custom` version, which does not defer checks but sends each message twice. Each party runs with 16 computation threads and four communication threads. We set Orq's batch size parameter to 2^{16} for sorting experiments (to better manage memory) and use the default configuration for queries.

Protocol $\Pi^{A}_{\mathsf{CheckEqs}}$

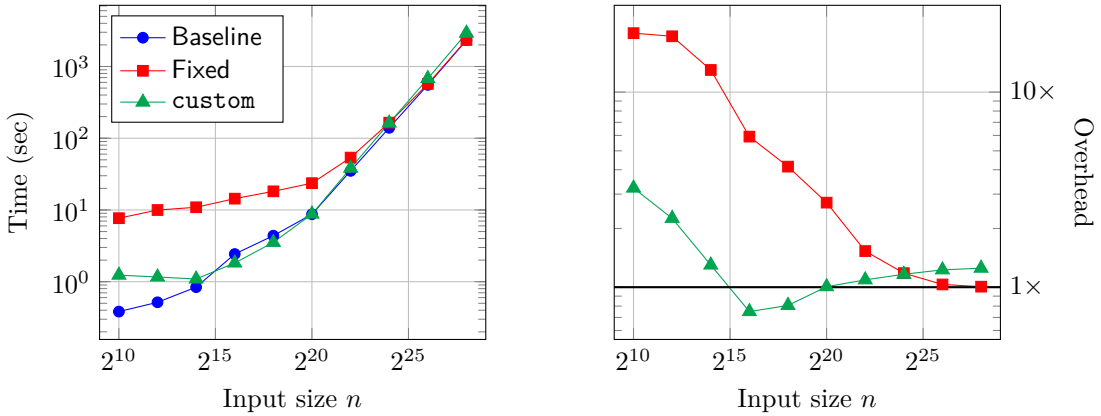
INPUT: $\forall a \in A$ with $a = [s \to r; h]$, digests d_a^H from P_h and d_a^R from P_r.

PROTOCOL: Let $n = |A|$. Interpret each 256-bit hash as an element of $\mathbb{F}_{2^{256}}$.

1. For each assignment $a \in A$, the hashing and receiving party use $\mathcal{F}_{\mathsf{ConsistentShare}}$ to secret-share their hashes.
2. Parties compute a local sum of the received shares and construct a canonically-ordered list $[\vec{d}] = [\vec{d}^H] \oplus [\vec{d}^R] \in \mathbb{F}^n_{2^{256}}$.
3. Call $[\vec{r}] \leftarrow \mathcal{F}_{\mathsf{Random}}(n)$ // Could be called in preprocessing.
4. Compute $[z] := \langle [\vec{d}], [\vec{r}] \rangle \in \mathbb{F}_{2^{256}}$ and open.

OUTPUT: If $z = 0$, accept. Otherwise, reject.

Figure 5: The protocol instantiating $\mathcal{F}^{A}_{\mathsf{CheckEqs}}$



(a) Absolute runtime

(b) Overhead compared to Baseline. Lower is better.

Figure 6: Performance of Quicksort under each 4PC implementation.

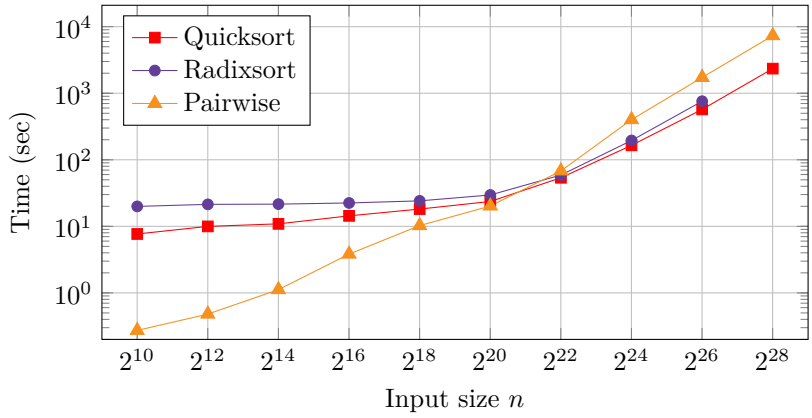


Figure 7: Performance of all sorting algorithms in the Fixed protocol. Radixsort runs out of memory at $n = 2^{28}$ due to permutation preprocessing.

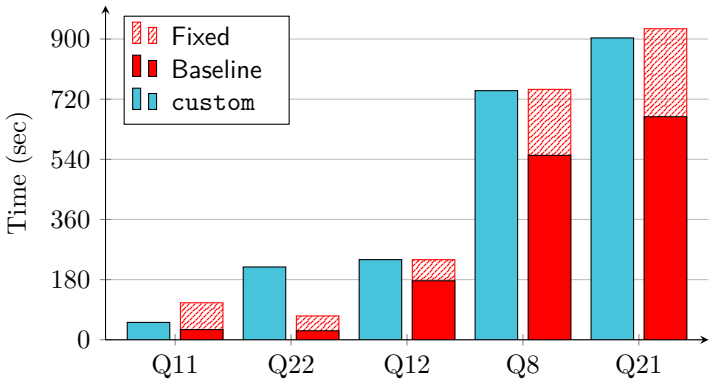


Figure 8: Performance on a subset of TPC-H queries

Sorting. Our first experiment runs Quicksort on a wide range of input sizes with all three protocols. Figure 6a shows the results. Baseline and `custom` perform similarly, and while Fixed imposes a large overhead at small sizes, it is quickly amortized away: at the largest data point, Fixed is only 0.6% slower than Baseline. Similarly, `custom` starts off faster than Fixed, but ends up being about 20% slower at the largest sizes, where its additional communication overhead becomes noticeable. Figure 6b plots the overhead of Fixed and `custom` relative to Baseline.

In Figure 7, we benchmark three of ORQ's sorting algorithms under Fixed. The pairwise sorting network [Par92] has an asymptotic runtime of $O(n \log^2 n)$ and operates strictly in the circuit model. Therefore, it does not require any additional malicious checks and performs quite well on inputs smaller than about 2^{20}. Past this point, Quicksort and Radixsort are largely equivalent and faster than the pairwise network. For more details on why malicious checks are necessarily within oblivious sorting algorithms, see Appendix B.

Somewhat surprisingly, while the fix itself only imposes an *additive* logarithmic overhead on the execution, and the sorting network a *multiplicative* overhead, the latter still performs well across a wide range of input sizes. On large vectors, it is concretely only about 3× slower than Quicksort. We expect that with additional engineering effort, however, the asymptotically-optimal sorting algorithms could be made faster at most sizes.

Queries. Next, we evaluate each of the 4PC protocols on a subset of five TPC-H queries benchmarked in ORQ's experiments: two fast queries, two slow queries, and a median-runtime query. Each query runs at Scale Factor 1 in LAN. Figure 8 shows the results. Compared to Baseline, which entirely omits malicious checks, Fixed is $1.4 - 3.6\times$ slower. While Baseline previously demonstrated speedups of $1.4 - 8.1\times$ over `custom`, after the fix, the story is more nuanced. We observe that Fixed is up to $2.1\times$ slower (Q11), and $3.1\times$ faster (Q22), than `custom`. For the more complex queries (Q12, Q8, Q21), the runtime of Fixed is within 3% of `custom`. These results raise the possibility that deferred checks may not even be necessary in many cases.

6 Further Improvements

Improved Round Complexity. ORQ implements the `multiply` primitive exactly as written in the original paper of Dalskov et al., with separate (and thus sequential) calls to `INP` in a loop. The framework does not currently provide a way to parallelize independent communication rounds. Thus, despite Fantastic Four's multiplication protocol technically only requiring one round, ORQ's implementation currently incurs six, notwithstanding the (four-round) `custom` implementation.

In a similar vein, each call to $\Pi_{\mathsf{ConsistentShare}}$ in the Verifier protocol occurs sequentially, but these, too, could be combined into a single parallel round, with the final consistency check deferred to a single hash comparison at the end of the sharing phase. Optimizing round complexity is likely not a challenging task, but does require some software redesign. Here, we prefer a simpler implementation for which security is easier to verify. Precisely because these kinds of optimizations led to the very issue exploited by [BS26], we defer such improvements to future work.

Log-Depth Equality Circuit. Another method of performing secure hash comparison would be to execute a standard oblivious equality circuit, as discussed in Section 4.2. While ORQ provides such an operator (implemented via the `BSharedVector::bit_same` method), it exists at a level of the software stack that is difficult for us to access from within the Verifier protocol. However, it is worth further investigation to remediate these minor software design conflicts, benchmark such an implementation, and compare it to our linear-combination version. Notably, the standard equality circuit does not need random secret-shared values, so the `CommittedSeedsQueue` could be omitted, greatly simplifying the implementation. Our understanding is that this approach was taken by the authors of HPMPC [HKSW+25] when fixing their protocol.

Full Security Proof. We have only proposed functionalities and the necessary protocol instantiations for the malicious-secure batch check. Future work should formalize these arguments and prove the simulation security of our implementation.

7 Conclusion

Deferred malicious checks in the Fantastic Four protocol are important for performance and proper resource utilization. After years of use by the cryptographic community, recent work has shown a significant attack on privacy when a common batching optimization is used. In this report, we have described how we implemented a constant-round, random linear-combination check within the ORQ framework for secure analytics. We demonstrated that our new implementation achieves comparable performance, while leaving room for further systems-level optimizations. Additionally, we identify and fix further malicious-

security issues in Orq. We hope our work can provide a blueprint for further research into practical malicious-secure protocols.

Acknowledgements

We are grateful to Andreas Brüggemann for many indispensable discussions about the finer details of implementing their proposed fix, and for identifying bugs in the first versions of the random-share functionality $\mathcal{F}_{\mathsf{Random}}$. Mayank Varia provided initial guidance on the pitfalls of implementing malicious-secure protocols. Sam Buxbaum and Adam Godel reviewed the implementation and many drafts of this report, and Muhammad Faisal prototyped an initial version of `CommittedSeedsQueue`.

TikZ diagrams were created with the assistance of generative artificial intelligence.

References

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016. doi:10.1145/2976749.2978331.

[BBM+25] Eli Baum, Sam Buxbaum, Nitin Mathai, Muhammad Faisal, Vasiliki Kalavri, Mayank Varia, and John Liagouris. Orq: Complex analytics on private data with strong security guarantees. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, pages 802–833, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3731569.3764833.

[BK82] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Comput.*, 31(3):260–264, March 1982. doi:10.1109/TC.1982.1675982.

[BS26] Andreas Brüggemann and Thomas Schneider. When trying to catch cheaters breaks the mpc: Breaking and fixing delayed consistency checks in trident, fantastic four, swift, and quad. In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology – EUROCRYPT 2026*, pages 457–487, Cham, 2026. Springer Nature Switzerland. doi:10.1007/978-3-032-25324-8_16.

[DEK21] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2183–2200. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov.

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014. doi:10.1145/2591796.2591861.

[HKI+13] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC 12*, volume 7839 of *LNCS*, pages 202–216. Springer, Berlin, Heidelberg, November 2013. doi:10.1007/978-3-642-37682-5_15.

[HKSW+25] Christopher Harth-Kitzerow, Ajith Suresh, Yongqin Wang, Hossein Yalame, Georg Carle, and Murali Annavaram. High-throughput secure multiparty computation with an honest majority in various network settings. *PoPETs*, 2025(1):250–272, January 2025. doi:10.56553/popets-2025-0015.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020. doi:10.1145/3372297.3417872.

[KNOS25] Alexander Kyster, Frederik Huss Nielsen, Sabine Oechsner, and Peter Scholl. Rushing at SPDZ: On the practical security of malicious MPC implementations. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 2491–2508. IEEE Computer Society Press, May 2025. doi:10.1109/SP61157.2025.00176.

[Par92] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 02(02n03):205–211, 1992. doi:10.1142/S0129626492000337.

A Deferred Protocol Specifications

Below, we provide protocols implementing the remaining functionalities defined in the main text.

B Malicious Sorting

In the standard circuit model of MPC, malicious checks can always be deferred until the end of a computation (or, at least, until the first outputs are revealed). This is the model considered by [BS26]; they argue that the amortized cost of the check is minimal, since it only needs to run once per computation. However, more precisely, the check must be performed before revealing *any* secret-shared values to the computing parties.

Orq breaks the strict circuit model of MPC with its implementations of Quicksort (following the shuffle-then-sort paradigm [HKI+13]) and shuffle (and, thus, Radixsort, as well). These algorithms reveal masked values, so we must run an additional malicious check in each iteration (of which, under most workloads, we only expect there to be $O(\log n)$).

Malicious parties can deviate from the protocol only in manners equivalent to introducing additive errors on the wires of a computation [GIP+14]. (Sorting takes place under boolean sharing, so this is an arbitrary XOR offset.) In Quicksort, comparisons may be repeated, since some pivots are reused across the control flow. Thus, for an arbitrary element x compared with pivot p, the adversary could allow $x < p$ to execute normally once, but introduce an offset δ to learn $x < p \oplus \delta$ the second time. Then, the adversary learns, say, whether $p < x < p \oplus \delta$ (as well as $p < p \oplus \delta$), leaking sensitive information about both p and x. We defer to future work whether this leakage can be extended into a concrete attack, and, if not, whether more lightweight malicious checks are sufficient within Quicksort. In our implementation, we require a full malicious check before every call to `open` per iteration of Quicksort.

This issue may extend to Orq's implementation of Radixsort, but in a slightly more nuanced way, as the honest-majority shuffle protocol involves opening a masked permutation. If a malicious party manipulates the generation of each bitwise permutation (within `radix_sort_body`), opening this masked permutation may have unexpected security implications. To avoid this potential issue, our new malicious check executes at the end of each round of Radixsort, giving an $O(\ell)$ additive overhead (for ℓ-bit shares). We do not yet know whether it is sufficient to run a single check at the end of the sorting protocol.

Protocol Π_{Random}

PARAMETERS: A keyed hash function $H(\cdot)$ with key domain $\mathbb{K}$, seed domain $\mathbb{S}$, and output ring $\mathcal{R}$. In our implementation, $\mathbb{K} = \mathbb{S} = \mathcal{R} = \mathbb{F}_{2^{256}}$.

---

subroutine REPOPULATE(batch size b): All parties execute the following.

1. Sample fresh keys $\vec{k} \xleftarrow{\$} \mathbb{K}^b$ and seeds $\vec{s} \xleftarrow{\$} \mathbb{S}^b$.
2. Commit to seeds: generate $\vec{c}$, with $\forall i \in [1, b] : c_i := H_{k_i}(s_i)$
3. Generate a batch commitment $M := H(\vec{c})$
4. Send $(\vec{c}, M)$ to all other parties and store $\vec{k}, \vec{s}$ in local key and seed queues, respectively.
5. When party P_j receives $(\vec{c}_k, M_k)$ from party P_k, abort if $M_k \neq H(\vec{c}_k)$. Then, send M_k to all other parties besides P_j (say, P_g, P_h)
6. Confirm batch consistency: When P_g (resp. P_h) receives M_k from P_j (call this M_k^j), abort if P_g's received batch commitment differs, $M_k^j \neq M_k^g$.

If the subroutine did not abort, parties append received commitments $\vec{c}_k$ to the local commitment queue Q_k and discard M_k, for all other parties P_k.

---

subroutine NEXTPRG(group $g \subseteq \{0, 1, 2, 3\}$): When called by party P_i, if $i \notin g$, P_i deletes the first commitment from each of its local queues $Q_k : \forall k \in g$ and returns. (This step is required to keep all parties' commitment queues in sync.)

Otherwise, P_i pops the next key and seed from its own queues and exchanges both with all other parties $P_j : j \in g$. Upon receipt of s_j, k_j from party P_j, P_i pops its next commitment c_j from local queue Q_j and aborts if $c_j \neq H_{k_j}(s_j)$. Parties construct a joint seed $s := \bigoplus_{j \in g} s_j$ and return $\mathsf{PRG}(s)$.

INPUT: $n \in \mathbb{Z}$, the number of secret-shared random values to generate

PROTOCOL: Initialize a secret-shared vector $[\vec{r}] \in \mathcal{R}^n$. For all size-3 subsets of parties $g \subset \{0, 1, 2, 3\}$:

1. Let $G \leftarrow$ NEXTPRG(g) and P_x be the excluded party (with $x \notin g$).
2. All parties besides P_x set $[r_i]_x \leftarrow G(i) : \forall i \in [1, n]$

OUTPUT: Parties return $[\vec{r}]$, a secret-shared vector of length n.

Figure 9: Protocol implementing the random-share generation functionality, which we instantiate with `CommittedSeedsQueue` in our implementation. Subroutine REPOPULATE should be called infrequently, with large batch sizes, to amortize the cost of generating, sharing, and verifying commitments.

Protocol $\Pi_{\mathsf{ConsistentShare}}$

SETUP: Use $\mathcal{F}_{\mathsf{Random}}$ to initialize a shared PRG seed for each subset of parties.

INPUT: P_i holds a message m

PROTOCOL: Arbitrarily assign the other three parties P_j, P_g, P_h;
e.g., $(j, g, h) := i + (1, 2, 3) \bmod 4$.

1. All parties set share $[m]_i = 0$
2. Parties P_i, P_g, P_h sample a random share $[m]_j$ from their shared PRG.
3. Parties P_i, P_j, P_g sample a random share $[m]_h$ from their shared PRG.
4. Party P_i computes $[m]_g = m - [m]_j - [m]_h$ and sends to P_j, P_h.
5. Parties P_j, P_h receive $[m]_g$ and exchange the received share[a] with each other. If the values do not match, abort.

OUTPUT: Parties output the secret sharing $[m]$

[a] Alternatively, parties could exchange a hash $H([m]_g)$ to minimize bandwidth. However, in our Verifier protocol, parties only operate on shared hashes, so further hashing provides no bandwidth improvement.

Figure 10: Our protocol instantiating $\mathcal{F}_{\mathsf{ConsistentShare}}$