Off-Key: Inconsistent MAC Keys Compromise Privacy in SPDZ-Family MPC

Marcel Keller[1], Sela Navot[2] and Nirvan Tyagi[3]

[1] CSIRO Technology, Australia
[2] Artemis Software Works, USA
[3] University of Washington, USA

Abstract. SPDZ-family protocols for maliciously secure multi-party computation protect secret-shared values with message authentication codes (MACs) under a secret-shared global key. Their security requires that all values and preprocessing used together be authenticated under the same key. We present Off-Key, a privacy attack that exploits violations of this requirement. Depending on which specific MAC check protocol is used, a corrupted party can either recover the value of a shared secret from a single authentication check or test a guess through selective failure by modifying a MAC share and observing whether a masked check succeeds. While this attack does not compromise the theoretical security guarantees of the protocols, we demonstrate this key-consistency vulnerability in evaluated versions of MP-SPDZ, SCALE-MAMBA, and FANNG-MPC, which allow a corrupted party to use different keys across batches, threads, or process runs. To help implementations avoid this class of attacks, we identify the state that enforces key consistency in representative preprocessing protocols and show how preserving and reusing it prevents the attack.

Keywords: multi-party computation

1 Introduction

Secure multi-party computation (MPC) lets several parties jointly evaluate a function of their private inputs while learning nothing beyond the result [GMW87]. Applications in industry and government include private model training [XZ24], public-health analytics [AG21], collaborative statistical analysis [BKK+16; ETZS26], and distributed transaction signing [Lin23]. This practical interest is also reflected in ongoing standardization efforts [ITU21; ISO23; ISO24; GPP+26; BP26]. These developments make it timely to investigate how to preserve MPC's security guarantees in real implementations, and how attackers can exploit gaps between theory and practice.

SPDZ [DPSZ12; DKL+13] is a widely used family of MPC protocols in the malicious security setting, which requires privacy and correctness even when corrupted parties deviate arbitrarily from the protocol. SPDZ evaluates arithmetic circuits over a large finite field using *authenticated secret sharing*: each shared value carries a shared message authentication code (MAC) under a secret-shared global key. Its fast online computation consumes authenticated correlated randomness supplied by preprocessing. The parties check MACs on opened values, including masked intermediate values whose underlying secrets remain hidden.

E-mail: Marcel.Keller@csiro.au (Marcel Keller), selanavot@gmail.com (Sela Navot), tyagi@cs.washington.edu (Nirvan Tyagi)

SPDZ's security proofs require all secret-shared values and preprocessing used together to be authenticated under the same key [DPSZ12; DKL+13]. We show that a corrupted party can exploit a known additive offset between keys used in the same computation to break privacy. SPDZ has many variants, and the precise leakage depends on the specific MAC check used. With SPDZ's unmasked check [DKL+13], the adversary can recover an entire secret-shared value, such as the honest party's input, from a single check. If one uses the masked MAC check from *Rushing at SPDZ* [KNOS25], which was introduced to mitigate another implementation pitfall, inconsistent keys still permit a *selective-failure attack*: by modifying a MAC share, the adversary can make the check's success or failure reveal whether a guess of the secret is correct.

These attacks do not contradict SPDZ's formal security guarantees: they exploit implementations that fail to preserve the key consistency required by the protocol. However, maintaining a consistent authentication key in an implementation is not trivial. No party knows the global authentication key, and the MACs on hidden values are themselves secret-shared, so a party cannot simply inspect a value to determine which key authenticates it. The online portion of SPDZ consumes correlated randomness, and its security proof assumes that this randomness is authenticated under the same key. The consistency requirement is thus pushed to the preprocessing protocol that generates the correlated randomness. Preprocessing itself is complex because implementations support different approaches, including homomorphic encryption and oblivious transfer, with different tradeoffs [DPSZ12; KOS16; KPR18].

Another complication arises from the high computational cost of SPDZ preprocessing protocols, which makes it useful to generate preprocessing in parallel, to generate new batches on demand when more material is needed, and to resume computations after a server restart without discarding previously authenticated values. Combining values from these different preprocessing executions requires them to be authenticated under the same key. Note that having each party save its key share does not prevent a malicious party from supplying a different share when preprocessing restarts.

All three frameworks we examined failed to enforce key consistency. MP-SPDZ [Kel20], SCALE-MAMBA [ACC+21], and FANNG-MPC [AAG+25] all had computation paths that started preprocessing multiple times, with each party supplying its share of the authentication key. A corrupted party could therefore supply different key shares across sessions, enabling the attack when their outputs were used together. We demonstrate the attack with proofs of concept in all three frameworks.

The fix lies in the preprocessing protocols themselves; they already enforce the key consistency needed by their security proofs within each execution. To provide this guarantee, they establish internal cryptographic state that binds each party to its key share. We refer to this state as the *key binding*, whose form depends on the individual preprocessing protocol and is specified through its initialization and authentication steps. Each party holds part of this state, and as long as the honest parties continue to use their parts of the key binding, a malicious party cannot silently change the key used for authentication [DPSZ12; DKL+13; KOS16; KPR18].

We identify the state that constitutes the key binding for several preprocessing protocols. Once this state is properly identified at the protocol level, implementations must preserve each party's part across batches, threads, and restarts and reuse it in all preprocessing executions whose outputs may be used together.

2 Related Work

SPDZ Implementation Issues. *Rushing at SPDZ* [KNOS25] most closely resembles our work. Its key observation is that a malicious party can learn the authentication key at the cost of failing an authentication check, allowing it to forge an output in another thread

running in parallel before the failure is detected in vulnerable SPDZ implementations. To mitigate this pitfall, the authors introduce a masked MAC check that prevents this key leakage. We show that this change also reduces our attack from full input recovery to selective failure when preprocessing keys are inconsistent.

Other MPC Security Issues. The security of other MPC implementations has also received attention. In the honest-majority setting, Brüggemann and Schneider show that delayed consistency checks can reveal intermediate wire values even though the cheating eventually causes a check to fail [BS26]. Related issues arise at the interface between garbling schemes and circuit compilers: Nieminen and Schneider show that a circuit can compute the intended function while violating implicit restrictions required for secure evaluation [NS23].

A different approach to evaluating protocol and implementation security trains machine learning models to compare what can be inferred from real and ideal executions. MPCGuard [LDS+25] applies this approach using neural networks, identifying leakage in 12 of 29 tested protocol implementations in the semi-honest setting. DT-SIM by Bates and Near [BN24] uses decision trees to detect deliberately introduced privacy flaws in GMW implementations.

Selective Failure in Shuffling. SPDZ-like circuits can be extended with permutation gates on secret-shared data. *Secret-Shared Shuffle with Malicious Security* [SYB+24] shows that earlier proposals, including Laud's shuffle [Lau21] and the related Clarion protocols [EB22], admit protocol-level attacks: authentication checks can reveal whether a guess about the hidden permutation is correct. In Section 7, we apply Off-Key to this setting, showing how inconsistent preprocessing keys compromise privacy.

3 Background: MPC and SPDZ

Secure multi-party computation (MPC) lets parties jointly evaluate an agreed function of their private inputs [GMW87]. We consider *malicious security with abort*, which protects privacy and correctness against arbitrary deviations by corrupted parties [GL05; IKK+11]. Informally, privacy requires that the adversary learn nothing about honest inputs beyond what its own inputs and prescribed outputs reveal. Correctness requires honest parties to accept only the agreed function's output on honest inputs and the adversary's chosen inputs.

We do not require fairness or guaranteed output delivery. The adversary may prevent honest parties from receiving their outputs even after obtaining its own.[1] However, privacy must hold even on abort: whether the protocol completes or aborts must not reveal information about honest inputs beyond what the adversary could learn from its chosen inputs and prescribed output.

For simplicity, we restrict our discussion to two parties P_0 and P_1, with at most one corrupted party. The parties evaluate an arithmetic circuit over a finite field $\mathbb{F}$ whose order is exponential in the security parameter. In the attacks, P_0 is corrupted and P_1 is honest; we denote their inputs by x and y, respectively.

3.1 Multi-Party Computation on Secret Shares

A simple approach for secure multi-party computation, albeit insufficient for malicious security, is to evaluate arithmetic circuits over additive shares [GMW87]. At each wire,

[1] In general, neither fairness nor guaranteed output delivery is achievable without an honest majority [Cle86].

the parties hold an additive sharing $[x]$ of its value $x \in \mathbb{F}$, with local shares satisfying

$$x = [x]_0 + [x]_1.$$

To input a fixed x, its owner samples one share uniformly, sends it to the other party, and retains the complementary share. Note that each additive share alone reveals no information about x. Addition gates and multiplication by a public constant require no communication, since each party can apply the operation to its own shares; addition of a public constant requires just one party to add it to its local share.

Multiplication is more complicated, but can be done using Beaver's trick [Bea91]. It uses preprocessed correlated randomness in the form of a fresh secret-shared triple $([a], [b], [c])$, where a, b are independent uniform field elements, independent of the inputs and hidden from either party, and $c = ab$. The parties locally form $[e] = [x] - [a]$ and $[d] = [y] - [b]$, then exchange their shares to open $e = x - a$ and $d = y - b$. They compute shares of the product locally as

$$[xy] = [c] + e[b] + d[a] + ed.$$

Since the triples do not depend on the inputs, they can be preprocessed in batches before the computation. To reconstruct an output, the parties exchange its shares and sum them.

However, this protocol does not ensure correctness against a malicious party. For example, after P_1's input y has been shared, P_0 can add Δ to its own share. This changes the represented input to $y + \Delta$ and hence can change the result of the computation. We therefore need to ensure that a malicious party uses correct and consistent shares throughout the protocol.

3.2 SPDZ: From Secret Shares to Malicious Security

To achieve malicious security, SPDZ augments each shared value with a shared message authentication code (MAC) [DPSZ12; DKL+13]. The parties hold additive shares of a uniformly sampled global MAC key α, and a value x is authenticated by a tag $m_x = \alpha x$:

$$\alpha = [\alpha]_0 + [\alpha]_1, \qquad m_x = [m_x]_0 + [m_x]_1 = \alpha x.$$

We write $\langle x \rangle^\alpha$ for the authenticated sharing, with local state $\langle x \rangle_i^\alpha = ([x]_i, [m_x]_i)$ at P_i. The key share $[\alpha]_i$ is held separately and used for all values in the computation. A sharing is considered valid under α when its reconstructed tag equals α times its reconstructed value.

As above, addition, multiplication by a public constant, and addition of a public constant can all be performed locally.

Sharing a party's input x is more involved because it requires a secret-shared tag $[m_x]$, where $m_x = \alpha x$, but neither party knows α. For this, preprocessing supplies input tuples, each consisting of an authenticated sharing $\langle r \rangle^\alpha$ of a uniform mask r, with r also revealed privately to the input owner P_i. To input x, P_i consumes a fresh tuple and broadcasts $\epsilon = x - r$, and both parties locally form $\langle x \rangle^\alpha = \langle r \rangle^\alpha + \epsilon$.

Similarly, multiplication uses a fresh authenticated Beaver triple $(\langle a \rangle^\alpha, \langle b \rangle^\alpha, \langle c \rangle^\alpha)$ from preprocessing, with $c = ab$. Following Section 3.1, the parties open the masked differences $e = x - a$ and $d = y - b$, retain their tag shares for MAC checking, and locally compute $\langle xy \rangle^\alpha$ without opening xy. Figure 1 presents the preprocessing functionality in more detail.

It is straightforward to show that modifying a secret's value while preserving validity requires guessing α correctly, which happens only with negligible probability. However, checking validity efficiently without revealing α requires care.

SPDZ preprocessing functionality

Initialization with MAC-key shares.

1. Receive $[\alpha]_i$ privately from each P_i, $i \in \{0, 1\}$.
2. Store $\alpha = [\alpha]_0 + [\alpha]_1$ for all subsequent calls.

Initialize once; the following operations may be called repeatedly.

Random(): **authenticated random value.**

1. Sample fresh $r \xleftarrow{\$} \mathbb{F}$.
2. Return $\langle r \rangle_i^\alpha$ to each P_i.

InputTuple$_k$(): **mask for** P_k **(**$k \in \{0, 1\}$**).**

1. Sample fresh $r \xleftarrow{\$} \mathbb{F}$.
2. Return $\langle r \rangle_i^\alpha$ to each P_i; also return r privately to P_k.

Triple(): **multiplication triple.**

1. Sample fresh, independent $a, b \xleftarrow{\$} \mathbb{F}$ and set $c = ab$.
2. Return $(\langle a \rangle_i^\alpha, \langle b \rangle_i^\alpha, \langle c \rangle_i^\alpha)$ to each P_i.

Figure 1: SPDZ preprocessing functionality.

Opening and MAC Checking. Malicious security requires checking the MAC of every opened value, including intermediate values and circuit outputs. Recall that opening $\langle v \rangle^\alpha$ reveals v, while its tag m_v and the key α remain secret-shared, and therefore checking that $m_v = \alpha v$ requires a small MPC protocol.

To do that, the parties use the public value v to locally compute plain additive shares of $m_v - \alpha v$ as

$$\sigma_i = [m_v]_i - [\alpha]_i v.$$

The original, *unmasked MAC check* commits to and then reveals σ_0 and σ_1, aborting if a commitment opening fails or their sum is nonzero [DKL+13].

Rushing at SPDZ [KNOS25] identifies that opening an incorrect value v can reveal α in this MAC check, making secure parallel implementations challenging. The paper therefore introduces a *masked MAC check* that uses a fresh preprocessed triple to open $\mu(\sigma_0 + \sigma_1)$ for a fresh, secret, uniform $\mu \in \mathbb{F}$, and aborts if the result is nonzero.

For either variant, the parties can batch many opened values into a single MAC check. They check a random linear combination of the opened values and their tags, using public random coefficients jointly sampled using commit-and-reveal after the opened values are fixed [DKL+13].

Figure 2 presents the SPDZ online protocol with both MAC-check variants. During circuit evaluation, intermediate openings accumulate for a batch check. Whenever a circuit output is ready, the parties check all pending openings before revealing the output, then open the output, and finally check its validity before accepting it.

SPDZ Preprocessing. Protocols for generating the authenticated masks and triples use techniques such as homomorphic encryption with proofs of plaintext knowledge [DPSZ12; DKL+13; KPR18], oblivious transfer on bit decompositions of field elements [KOS16], and vector oblivious linear evaluation [RS22].

As discussed in the introduction, our attack exploits inconsistent MAC keys across preprocessing sessions whose outputs are used together. Our fix uses a key-binding abstraction to specify what state must persist across these sessions. The form of this state depends on the preprocessing protocol. We therefore describe representative constructions alongside

SPDZ Online Protocol

Initialization with preprocessed shares. Each P_i, $i \in \{0,1\}$, holds:

- Its MAC-key share $[\alpha]_i$.
- A list of Beaver-triple shares $(\langle a\rangle_i^\alpha, \langle b\rangle_i^\alpha, \langle c\rangle_i^\alpha)$, with $c = ab$.
- For each input owner P_k, a list of input-tuple shares $\langle r\rangle_i^\alpha$; only P_k also receives the corresponding r.
- An initially empty list L_i of pending MAC checks.

$\mathsf{Input}_k(x_k)$**: input from** P_k **(**$k \in \{0,1\}$**).**

1. Consume an input tuple $\langle r\rangle^\alpha$, with uniform r known only to P_k.
2. P_k sends $\epsilon = x_k - r$ to P_{1-k}; both parties locally form $\langle x_k\rangle^\alpha = \langle r\rangle^\alpha + \epsilon$.

Local arithmetic. For a public constant u, each P_i computes:

	$[z]_i$	$[m_z]_i$
$\mathsf{Add}(\langle x\rangle^\alpha, \langle y\rangle^\alpha)$	$[x]_i + [y]_i$	$[m_x]_i + [m_y]_i$
$\mathsf{MulConst}(\langle x\rangle^\alpha, u)$	$u[x]_i$	$u[m_x]_i$
$\mathsf{AddConst}(\langle x\rangle^\alpha, u)$	$[x]_i + (1-i)u$	$[m_x]_i + u[\alpha]_i$

$\mathsf{Mul}(\langle x\rangle^\alpha, \langle y\rangle^\alpha)$**: multiplication.**

1. Consume a fresh triple $(\langle a\rangle^\alpha, \langle b\rangle^\alpha, \langle c\rangle^\alpha)$, with independent uniform a, b and $c = ab$.
2. Obtain $e \leftarrow \mathsf{Open}(\langle x\rangle^\alpha - \langle a\rangle^\alpha)$ and $d \leftarrow \mathsf{Open}(\langle y\rangle^\alpha - \langle b\rangle^\alpha)$.
3. Compute locally and return $\langle xy\rangle^\alpha = \langle c\rangle^\alpha + e\langle b\rangle^\alpha + d\langle a\rangle^\alpha + ed$.

$\mathsf{Open}(\langle v\rangle^\alpha)$**: internal opening with a deferred check.**

1. Exchange value shares and reconstruct $v = [v]_0 + [v]_1$.
2. Each P_i appends $(v, [m_v]_i)$ to L_i. Return v for use inside the protocol; its MAC has not yet been checked.

$\mathsf{Output}(\langle v\rangle^\alpha)$**: reveal one value.**

1. Run $\mathsf{Check}()$ on all pending intermediate openings.
2. Reconstruct $v \leftarrow \mathsf{Open}(\langle v\rangle^\alpha)$, then run $\mathsf{Check}()$ on this output.
3. Return v to both parties only if the check accepts. Further operations may follow, with L_0, L_1 now empty.

$\mathsf{Check}()$**: check the pending openings.**

1. Parse L_i as t pairs $(v_j, [m_{v_j}]_i)$, $j = 1, \ldots, t$. If $t = 0$, return success.
2. Jointly sample a fresh uniform $\rho \in \mathbb{F}$ using commit-and-reveal.
3. Each P_i locally computes the plain additive share

$$\sigma_i = \sum_{j=1}^{t} \rho^{j-1}[m_{v_j}]_i - [\alpha]_i \sum_{j=1}^{t} \rho^{j-1} v_j.$$

4. **Unmasked:** each P_i sets $z_i = \sigma_i$.

4. **Masked:** consume a fresh triple and discard its MAC shares, leaving $([p], [\mu], [q])$ with independent uniform p, μ and $q = p\mu$. Exchange $\delta_i = \sigma_i - [p]_i$ and reconstruct $\delta = \delta_0 + \delta_1$. Each P_i sets $z_i = \delta[\mu]_i + [q]_i$.

5. Commit and reveal z_0, z_1; abort if a commitment opening fails or $z_0 + z_1 \neq 0$. Otherwise, clear L_i and return success.

Figure 2: Two-party SPDZ online computation [DPSZ12; DKL+13] with either the original unmasked MAC check (solid box) or the masked check from *Rushing at SPDZ* [KNOS25] (dashed box).

the fix in Section 5. There, we explain how each construction generates authenticated correlated randomness, identify its key binding, and show how authentication uses it.

4 The Off-Key Attack

4.1 Warm-Up: Selective Failure from a Blind MAC-Check Oracle

As a way to introduce the attack, consider a validity oracle that returns only whether the submitted shares are valid. That is, it reports whether the reconstructed MAC passes the validity check under the reconstructed key. More concretely, each party P_i provides a value share $[y]_i$, a MAC share $[m_y]_i$, and a MAC-key share $[\alpha]_i$. The oracle checks whether

$$[m_y]_0 + [m_y]_1 \stackrel{?}{=} ([\alpha]_0 + [\alpha]_1)([y]_0 + [y]_1),$$

returning one if the equality holds and zero otherwise.

Suppose the adversary P_0 receives the authenticated share $\langle y \rangle_0^\alpha = ([y]_0, [m_y]_0)$ of a possibly low-entropy secret y, together with its MAC-key share $[\alpha]_0$. To test whether y equals a guess g, P_0 keeps $[y]_0$ fixed, adds g to $[m_y]_0$, and adds 1 to $[\alpha]_0$. The honest party submits its original shares.

Together, the submitted value shares still reconstruct to y, but the MAC now reconstructs to $\alpha y + g$ and the key to $\alpha + 1$. The oracle therefore returns one exactly when

$$\alpha y + g = (\alpha + 1)y \quad \Longleftrightarrow \quad g = y.$$

Its one-bit response reveals whether the guess was correct.[2]

Standard SPDZ does not provide this oracle directly, as the protocol only checks the validity of opened values (Figure 2). It therefore does not directly check an unopened secret y, and checking a guess of an opened secret reveals no additional information as, well, the secret is already opened. However, we show how implementations can implicitly allow an adversary to check the validity of modified shares of an unopened value, and thereby test a guess as described above.

4.2 Blind Authentication via SPDZ Multiplication

The key observation is that SPDZ also checks intermediate values opened during the computation. For example, a multiplication opens a masked value such as $d = y - b$, where b is a secret, uniformly random mask independent of y. Because b is uniform, d reveals nothing about y. Yet its MAC depends on y's MAC, and its authentication is checked while y itself remains hidden. This gives us something much closer to the blind MAC-check oracle from the warm-up.

Fortunately, this is still not sufficient for an attack. The remaining catch, on which SPDZ's security proof relies, is that the mask b is authenticated under the same key α as y. Suppose the adversary makes the same changes as in the warm-up, adding 1 to its key share and a guess g to its MAC share of y. Because the MAC on b is still αb, the MAC on d becomes $\alpha d + g$, while the check under $\alpha + 1$ expects $\alpha d + d$. The MAC equation therefore holds exactly when $g = d$. But d is already public and independent of y, so this reveals nothing about whether $g = y$.

[2] Note that the shift by 1 is arbitrary: for any nonzero $\Delta \in \mathbb{F}$, adding Δ to the key share and $\Delta \cdot g$ to the MAC share gives the same test. We use 1 for simplicity.

4.3 Off-Key Preprocessing Makes Blind Authentication Exploitable

The situation changes if the mask's MAC already matches the adversary's modified key. More concretely, suppose the adversary can arrange for the mask b to be authenticated under $\alpha+1$, while y is still authenticated under α. Before the adversary modifies its shares, the reconstructed MACs satisfy

$$m_y = [m_y]_0 + [m_y]_1 = \alpha y, \qquad m_b = [m_b]_0 + [m_b]_1 = (\alpha+1)b.$$

Now, the attack from the warm-up is reenabled. To test a guess g, the adversary modifies its shares of y and the key as before: it adds g to its MAC share of y and 1 to its key share, while keeping its value share of y and its shares of b unchanged. The honest party follows the standard protocol, leaving its shares unchanged.

When $d = y - b$ is opened, its reconstructed MAC differs from the MAC expected under $\alpha + 1$ by

$$\underbrace{([m_y]_0 + g) - [m_b]_0}_{P_0\text{'s MAC share on } d} + \underbrace{[m_y]_1 - [m_b]_1}_{P_1\text{'s MAC share on } d} - \underbrace{([\alpha]_0 + 1 + [\alpha]_1)d}_{\text{expected MAC on } d} = g - y.$$

Thus, d is correctly authenticated under $\alpha + 1$ exactly when $g = y$. Both MAC-check variants in Figure 2 therefore pass exactly when $g = y$, except with negligible probability. The same holds in the batched MAC-check setting, when d is checked together with openings correctly authenticated under $\alpha + 1$.

The attack above requires the input y to be authenticated under α and the multiplication mask b under $\alpha + 1$. This key mismatch can arise when an implementation combines material from separate preprocessing sessions without enforcing a common key. Suppose P_1 inputs y under α using a tuple generated in one preprocessing session. In another preprocessing session, P_0 supplies $[\alpha]_0 + 1$ while P_1 reuses $[\alpha]_1$, producing a mask b authenticated under $\alpha + 1$. Using this mask with the existing sharing of y and checking under $\alpha + 1$ enables the attack.

The standard SPDZ security proof assumes that all preprocessing material used together is authenticated under the same key [DPSZ12]. In Section 6, however, we show that the evaluated versions of three MPC frameworks allowed a malicious party to violate this requirement, enabling the attack.

4.4 Input Recovery with SPDZ's Unmasked Authentication Check

The blind MAC-check oracle captures only whether authentication succeeds. However, when the authentication keys are inconsistent, SPDZ's original unmasked MAC check in Figure 2 [DKL+13] exposes additional information, allowing the same attack to recover the honest input outright.

Recall that in the unbatched MAC check, when checking the validity of an opened value v, each party P_i computes

$$\sigma_i = [m_v]_i - [\alpha]_i v.$$

The parties commit to and then reveal σ_0 and σ_1, exposing $\sigma_0 + \sigma_1 = m_v - \alpha v$.

When y is authenticated under α, b under $\alpha + 1$, and the adversary has added g to its MAC share of y and 1 to its key share, applying this check to the opening $d = y - b$ from Section 4.3 reveals

$$\sigma_0 + \sigma_1 = g - y.$$

The adversary can then compute $y = g - (\sigma_0 + \sigma_1)$, even if the check subsequently aborts.

In a batched MAC check, the same recovery works if all openings other than d are correctly authenticated under $\alpha + 1$. Their MAC differences are zero, so the revealed sum

is $\sigma_0 + \sigma_1 = \lambda(g - y)$, where λ is d's public batching coefficient. When $\lambda \neq 0$, which holds except with negligible probability, the adversary recovers $y = g - (\sigma_0 + \sigma_1)/\lambda$.

The masked check from *Rushing at SPDZ* [KNOS25], introduced to address a different security pitfall in concurrent implementations, reduces our attack from direct recovery of y to selective failure when implementations allow off-key preprocessing, but it does not eliminate the selective-failure attack shown in Section 4.3.

4.5 Illustrative Examples

Example 1 (Multiplication with a zero input)**.** Consider a circuit $f(x, y) = xy$, where P_0 and P_1 supply x and y, respectively. This circuit can be seen as a sort of AND gate if one interprets zero as false and a nonzero output as true. If P_0 supplies $x = 0$, the prescribed output is always zero, so the circuit should reveal nothing about y.

Figure 3 shows our attack, in the setting where the input tuples are authenticated under α and the Beaver triple under $\alpha + 1$. The attack reveals information about the honest party's input, with the adversary using $g = 0$ in our terminology above.[3]

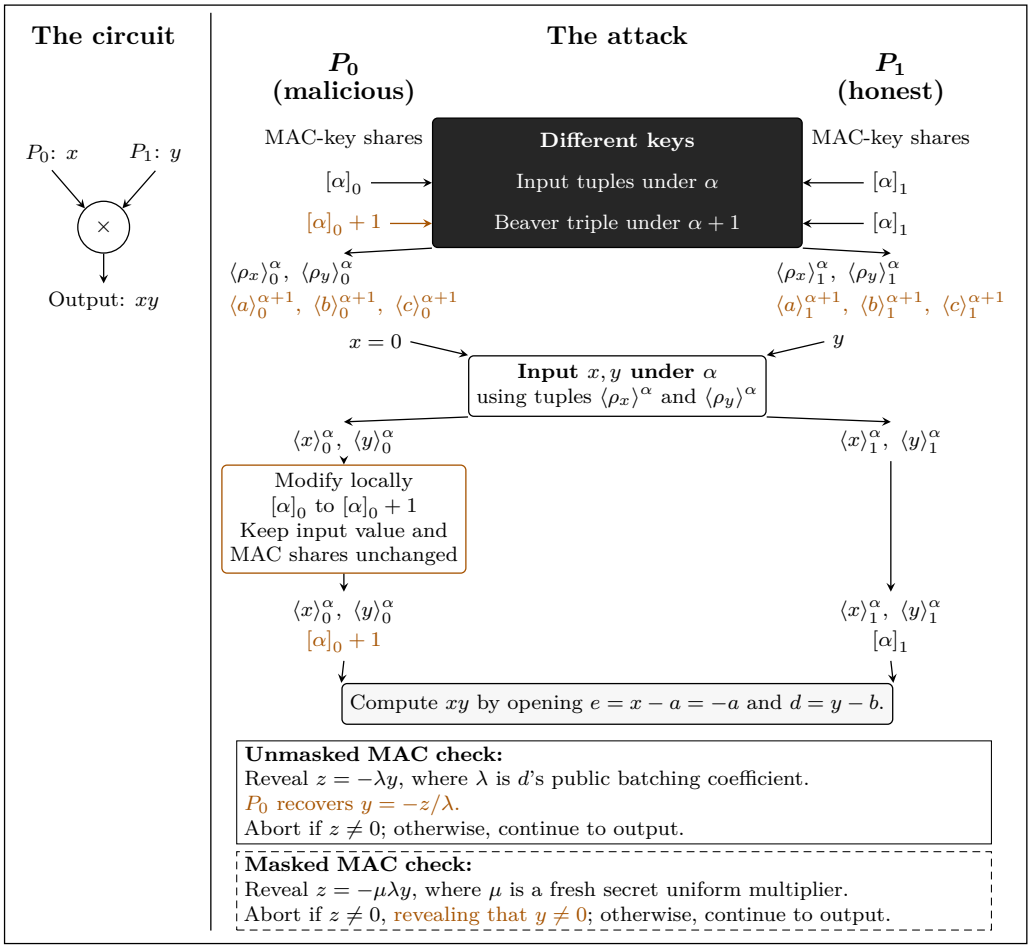

Figure 3: Left: the multiplication circuit $f(x, y) = xy$. **Right:** the attack with $g = x = 0$. The solid (unmasked) check recovers y; the dashed (masked) check tests the guess $y = 0$ through selective failure, both up to negligible error.

[3] We use this simplest instance to illustrate the attack, but its outcome can also be achieved within the security model. Under security with abort and no fairness [GL05], an ideal adversary can submit $x' = 1$, learn $y = f(1, y)$, and then allow P_1 to receive 0 when $y = 0$, or force an abort otherwise. Thus, this example alone does not establish a violation of malicious security. This motivates Example 2.

Example 2 (Private affine evaluation)**.** Consider $f(x;(y,r)) = xy + r$, where x is supplied by P_0 and (y,r) by P_1.

Figure 4 illustrates our attacks with $g = 1$ and $x = 1$. The input y is authenticated under α, while x, r, and the Beaver triple use $\alpha + 1$. After P_1 inputs y, P_0 adds 1 to its key share and its MAC share of y, making the MAC valid under the new key exactly when $y = 1$. With the masked MAC check, this tests $y = 1$ through selective failure; with the unmasked MAC check, it reveals y outright, in both cases except with negligible probability.

To see the privacy violation, let y be a uniform bit and r a uniform field element independent of y, with the result revealed only to P_0.[4] For every input x' chosen by an ideal adversary, $x'y + r$ is uniform and independent of y. Thus, in the ideal world, the adversary cannot guess y with probability greater than $1/2$. Our attacks recover y.

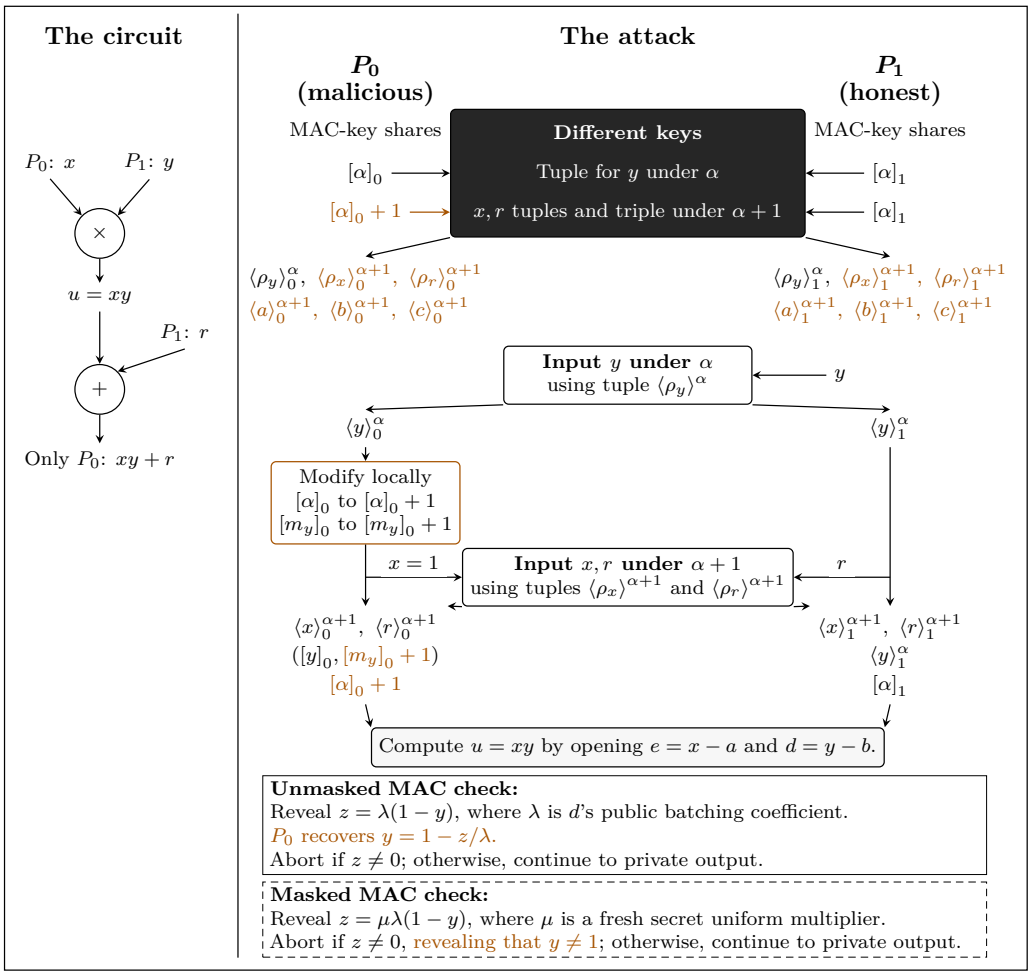

Figure 4: Left: private affine evaluation $f(x;(y,r)) = xy + r$, with output only to P_0. **Right:** the attack with $g = x = 1$. The solid (unmasked) check recovers y; the dashed (masked) check tests the guess $y = 1$ through selective failure, both up to negligible error.

[4] We use one-sided output because, for $y \neq 0$, a public output reveals x to P_1. This is the oblivious linear function evaluation (OLE) functionality used in TinyOLE [DGN+17]. To implement one-sided output, use a fresh uniform authenticated mask $\langle s \rangle^{\alpha+1}$, with s known only to P_0: publicly open and check $xy + r + s$, then let P_0 subtract s [DI05; AOR+19].

5 Mitigation: Maintaining Key Consistency Across Sessions

The attacks above do not break the preprocessing protocols: their existing initialization and authentication steps already enforce key consistency within an execution [DPSZ12; KOS16; KPR18]. The implementation mistake is to treat saved plaintext MAC-key shares as sufficient to preserve this guarantee across sessions.

Our contribution here is to identify the protocol-specific state that implementations must retain for key consistency. We call this state the *key binding*: together with the protocol's checks, it ties subsequent authentication to the initialized MAC key. It is established by the protocol's existing interactive setup.

Each party must preserve its part of this binding across preprocessing sessions whose outputs are used together. Independently regenerating it lets a malicious party supply a modified key share and bind new material to a different key, enabling the attack in Section 4.3. Implementations must still follow the protocol's usual abort rules, including aborting after a failed validity check. After such an abort, the parties must stop the computation and must not reuse the MAC key or any material authenticated under it.

5.1 Key Binding in Preprocessing Protocols

The binding takes a different form in each construction. We describe how representative protocols establish and use it. Table 1 summarizes their key bindings.

Somewhat Homomorphic Encryption. The original SPDZ preprocessing uses somewhat homomorphic encryption (SHE), which allows additions and a bounded number of successive multiplications on encrypted values. First, a separate setup process generates a public SHE encryption key pk (distinct from the SPDZ MAC key α) and gives each party a share of the decryption key sk. This one-time setup can be realized by a general MPC protocol [DPSZ12, Section 3]. Then, each party P_i chooses a random MAC-key share $[\alpha]_i$, encrypts it under pk, and proves knowledge of the plaintext and validity of the ciphertext. The parties add these encrypted shares to obtain a ciphertext C_α encrypting α. This aggregate ciphertext provides the key binding under the fixed encryption setup. Each party retains C_α, the public key pk, and its share of the decryption key sk for subsequent preprocessing [DPSZ12; DKL$^+$13].

With honest execution, authentication proceeds as follows. To generate an authenticated random mask r, each party chooses and encrypts a fresh share of r. Adding the encryptions gives an encryption of r, which they multiply by C_α to obtain an encryption of its MAC αr. The parties use a specialized MPC protocol with their shares of sk to convert this ciphertext into secret shares of the MAC without revealing it [DPSZ12; DKL$^+$13]. For an input tuple, r is also revealed privately to the input owner.

To generate an authenticated multiplication triple, the parties similarly choose and encrypt shares of random a and b. They authenticate each using C_α and obtain its MAC shares as above. They then compute an encryption of $c = ab$ homomorphically and obtain secret shares of c and its MAC αc in the usual way.[5] This describes the intended arithmetic; malicious decryption contributions can introduce additive errors, and malicious security additionally requires proofs on encrypted contributions, MAC checks, and a *sacrifice* step to verify that $c = ab$ [DPSZ12; DKL$^+$13]. We omit the details of these checks.

Later optimizations, including HighGear [KPR18] and TopGear [BCS19], improve the proofs of knowledge and encryption parameters while retaining C_α as the key binding. Their joint proofs certify aggregate ciphertexts rather than each party's ciphertext separately.

[5] To stay within SHE's bounded multiplication depth, the conversion to shares refreshes the encryption of c before multiplication by C_α. Thus one multiplication layer suffices at each step [DPSZ12; DKL$^+$13].

Linearly Homomorphic Encryption. Another approach is to use linearly homomorphic encryption (LHE), which lets a party add encrypted values and multiply an encrypted value by a scalar it knows. In Overdrive's LowGear construction [KPR18], each party P_i has its own encryption key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$. It chooses a random MAC-key share $[\alpha]_i$, encrypts it under pk_i, and sends the ciphertext to the other party with a proof of plaintext knowledge and ciphertext validity. These encrypted MAC-key shares provide the key binding under the fixed encryption setup. Each party retains the ciphertexts, both public keys, and its own decryption key for subsequent preprocessing [KPR18].

Authentication of a shared value x requires secret shares of its MAC αx. Expanding this product gives

$$\alpha x = [\alpha]_0[x]_0 + [\alpha]_1[x]_1 + [\alpha]_0[x]_1 + [\alpha]_1[x]_0.$$

Each party P_i can compute its local product $[\alpha]_i[x]_i$. The two remaining cross products involve shares held by different parties. For each cross product $[\alpha]_i[x]_j$ with $i \neq j$, party P_j multiplies the saved encryption of $[\alpha]_i$ by its value share $[x]_j$, subtracts an encryption of a fresh random mask, and sends the result to P_i for decryption. The decrypted value and the mask form additive shares of the cross product.[6] Adding these shares to the local products $[\alpha]_i[x]_i$ gives shares of the MAC αx. Applying this to fresh random value shares produces authenticated random masks; for an input tuple, the mask is also revealed privately to its owner.

To generate an authenticated multiplication triple, the parties use the same masked-product procedure on fresh random shares of a and b. They combine shares of the cross products $[a]_i[b]_j$ with the local products $[a]_i[b]_i$ to obtain shares of $c = ab$, then authenticate a, b, and c as above. Malicious security additionally requires proofs on encrypted contributions, batch consistency checks for authentication, MAC checks, and sacrifice. We omit the details of these checks.

Oblivious Transfer. MASCOT [KOS16] generates SPDZ preprocessing using oblivious transfer (OT). At initialization, each party P_i chooses a random MAC-key share $[\alpha]_i$ and decomposes it into ℓ bits $(\beta_{i,1}, \ldots, \beta_{i,\ell})$. For each bit β in this decomposition, the other party P_j offers two random *seeds*, labeled 0 and 1. A seed is a short random bit string from which a deterministic algorithm generates a sequence of pseudorandom field elements; anyone holding the same seed gets the same sequence. P_j keeps both seeds; OT lets P_i receive and keep only the seed labeled β, without revealing β to P_j or learning the other seed.

With honest execution, preprocessing generates an authenticated random mask r as follows. Each party samples a fresh random share $[r]_i$. The parties then need shares of the MAC αr. As in LowGear's expansion, each party P_i can compute $[\alpha]_i[r]_i$ itself, since it holds both factors. The product $[\alpha]_i[r]_j$ for $i \neq j$ involves factors held by different parties, so they use OT to share it. Consider one bit β of $[\alpha]_i$. The parties take the next unused element from each of that bit's seed sequences, where *next* is defined by a persistent sequence counter. Writing these elements as t_0 and t_1, P_j knows both, whereas P_i knows only t_β. P_j sends the correction $u = t_0 - t_1 + [r]_j$, and P_i computes

$$t_\beta + \beta u = t_0 + \beta[r]_j.$$

This value and P_j's $-t_0$ are additive shares of $\beta[r]_j$. Taking the corresponding weighted sum over the key bits gives shares of $[\alpha]_i[r]_j$. Doing this in both directions and adding each party's own $[\alpha]_i[r]_i$ gives shares of the MAC αr. The same seeds support later products with fresh masks, but P_i cannot change its choice bits: it does not know the unselected

[6] LowGear also randomizes ciphertext noise to hide the multiplying value and assumes enhanced CPA security to prevent leakage through its checks [KPR18].

seeds. The seeds retained by both parties therefore provide the key binding; both parties also retain the persistent sequence counter.

For an input tuple, the input owner samples a uniform mask r, distributes additive shares of it, and uses this procedure to obtain shares of its MAC αr. The owner retains r for its online input step. Authentication includes a masked batch MAC check to bind a malicious sender to its values.

To generate triples, the parties sample random shares of a and b and use fresh OTs to repeat the bit calculation above, replacing $[\alpha]_i$ by $[a]_i$ and $[r]_j$ by $[b]_j$. Adding each party's own product $[a]_i[b]_i$ to the resulting shares gives shares of $c = ab$. Before authentication, MASCOT forms random linear combinations of several candidate products to protect against leakage from malicious deviations. It then authenticates the resulting shares under the initialized key and uses sacrifice to check that $c = ab$. As above, we omit the details of these checks.

Vector Oblivious Linear Evaluation. Creating authenticated preprocessing material in SPDZ requires multiplying many secret-shared values by the same secret MAC key α. Vector oblivious linear evaluation (VOLE) is particularly suited to this task: it provides additive shares of products between one party's fixed secret scalar and a vector of values held by the other party. Using P_i's MAC-key share $[\alpha]_i$ as the scalar supplies shares of the cross products $[\alpha]_i[x]_j$ for $i \neq j$ and fresh value shares $[x]_j$. Combining these with the local products $[\alpha]_i[x]_i$, as above, gives shares of the MAC αx [RS22].

In the two-party specialization of Le Mans's authentication component [RS22], each party uses its MAC-key share as the scalar in a VOLE instance with the other party.[7] Each VOLE instance, by its own security guarantee, binds all products in a batch and later extensions to the same initialized scalar, and Le Mans's checks on the resulting authentication shares then bind the accepted values to the sum of those two scalars, which in our setting is the MAC key $\alpha = [\alpha]_0 + [\alpha]_1$. To maintain this guarantee across sessions, implementations must retain the key-binding state of both instances, whose concrete form depends on the underlying VOLE construction, and resume them with fresh correlations and the prescribed checks.

Ring-LPN-Based Silent Preprocessing. Abram and Scholl [AS22] reduce the communication cost of SPDZ preprocessing using maliciously secure silent preprocessing. The parties interact to generate short, correlated seeds, then expand them locally into large batches of authenticated input masks and multiplication triples.

At initialization, each party P_i inputs its MAC-key share $[\alpha]_i$ into the maliciously secure MPC used for seed generation. That MPC stores a protected representation of the input. To generate a fresh batch, it uses these same stored inputs in the cross-party products needed for authentication [AS22].

The key binding is the MPC state that fixes these inputs. For example, if the seed-generation MPC uses authenticated secret sharing, parties retain shares and MAC shares representing each initialized input, together with the MPC's own key binding. Fresh seed generation must reuse this state.

In the encryption-based protocols, the ciphertext bindings use the established encryption parameters and public keys. The parties also retain their decryption keys or key shares, but these keys alone do not fix the MAC key.

[7] While orthogonal to our discussion, we point out that Le Mans modifies SPDZ's preprocessing functionality and online protocol. Its native triples leave c unauthenticated and potentially unequal to ab, so the online phase authenticates c before use and checks the multiplications for consistency with a second, randomized circuit evaluation.

Table 1: Key bindings in representative SPDZ-family preprocessing protocols.

Protocol	Key binding
SPDZ [DPSZ12; DKL+13] HighGear [KPR18] TopGear [BCS19]	Initialized aggregate ciphertext C_α encrypting α, under the shared SHE setup
LowGear (LHE) [KPR18]	Accepted setup ciphertexts of $[\alpha]_i$ under each party's public key pk_i
MASCOT [KOS16]	Established OT correlations: sender seed pairs and the receiver's selected seeds, with choice bits encoding its MAC-key share; persistent sequence counter
Le Mans (VOLE) [RS22]	Initialized VOLE state in both directions binding the two MAC-key shares; concrete state depends on the underlying VOLE construction
Abram and Scholl (Ring-LPN silent preprocessing) [AS22]	Seed-generation MPC state that fixes the initialized MAC-key shares across batches; concrete state depends on the underlying MPC implementation

6 Attacking MPC Frameworks

We now show how a malicious party can create the key mismatch required by our attacks in three MPC frameworks. Using this key mismatch, we implement the full input-recovery attack of Section 4.4 and the weaker selective-failure attack of Section 4.3. We demonstrate both attacks against the circuits in Examples 1 and 2. The proof-of-concept code is available in our public GitHub repository.[8]

6.1 SCALE-MAMBA

SCALE-MAMBA is an MPC framework that combines a high-level programming language with a runtime for actively secure computation [ACC+21].

SCALE-MAMBA allows the parties to save authenticated values to disk and use them in a later execution, so a graceful shutdown and restart need not discard previously authenticated values [ACC+21]. For this to work securely, new preprocessing must use the same authentication key as the saved values. However, each new process reads its key share from a plaintext file and freshly encrypts it for FHE preprocessing. The accompanying proof of knowledge establishes validity within that session, but does not bind the key share across sessions. Consequently, a malicious party can change its local key share between executions without changing the saved values. When execution resumes, those values retain their original MACs, while fresh preprocessing uses the changed key [SCA22b].

Our proofs of concept exploit this mismatch by splitting the computation across two executions. In the first, P_1 supplies its input y, which is authenticated under α. The parties save their shares to disk and stop their processes. Before the second execution begins, P_0 adds 1 to its stored key share. The honest party's share remains unchanged, so new inputs and preprocessing use $\alpha + 1$, while the saved input y remains authenticated under α. To test a guess g, P_0 also adds g to its MAC share of y in the memory dump, leaving the value share unchanged. These two local file edits suffice for selective failure with the unmodified runtime.

The same mismatch also enables full input recovery. The saved MAC shares are left unchanged, and the unmasked MAC check lets P_0 recover y from the revealed check values

[8] https://github.com/selanavot/off-key-poc

and public batching coefficients [SCA22b], even if the check then fails.

SCALE-MAMBA has not been actively maintained since June 2022 [SCA22a], so no patch is planned. We have explicit permission from a former maintainer to publish this attack.

6.2 FANNG-MPC

FANNG-MPC is a fork of SCALE-MAMBA for privacy-preserving neural-network inference [AAG+25]. The evaluated full-threshold implementation inherits SCALE-MAMBA's cross-process key-loading behavior and unmasked MAC check [FAN25]. Our proofs of concept use the same two-run setup and local file edits as in Section 6.1.

The FANNG-MPC team gave explicit permission to publish our findings and proof-of-concept code, and at the time of this writing no patch has been applied.

6.3 MP-SPDZ

MP-SPDZ [Kel20] is a widely used MPC framework, supporting multiple protocols and security models through a common high-level programming interface. For SPDZ-family protocols, it saves each party's MAC-key share on disk for reuse in later executions, much like SCALE-MAMBA. Its runtime also supports multithreaded computation [Kel26b], so key consistency must hold both across executions and across threads.

The Attack. Before the fix, several preprocessing paths started from saved plaintext MAC-key shares, and each path could establish its own binding to the share it was given. This left MP-SPDZ vulnerable to the same cross-run attack as SCALE-MAMBA, described in Section 6.1 [MP-26a].

More interestingly, its multithreaded runtime also allowed an off-key attack within a single process. In its MASCOT implementation, each thread generated its own preprocessing using oblivious transfer (OT) and independently established an OT-based commitment to the party's MAC-key share. These commitments bound the key within each thread, but not across threads, allowing a malicious party to supply different key shares to different threads. Exchanging authenticated values through shared memory then brought together values and preprocessing under different keys [Kel26b].

Our proofs of concept authenticate P_1's input y under α in one thread, then pass it to a worker whose preprocessing uses $\alpha + 1$. In the affine circuit, the remaining inputs and private-output mask also use the worker's key. To test a guess g, the malicious runtime adds g to P_0's MAC share of y before multiplication. For full recovery, P_0 leaves that share unchanged and recovers y from the worker's unmasked MAC check [Kel26b]. Both parties execute the same MPC program; only P_0's runtime is modified.

Overview of the Fix. After disclosure, MP-SPDZ was patched in version 0.4.3 [MP-26b]. The patch makes the saved protocol setup the common starting point for preprocessing. The MAC-key share and its key binding are kept together in a protocol-specific file on disk, and later preprocessing starts from this saved setup.

For MASCOT, each party now saves its MAC-key share together with the OT sender seed pairs, selected receiver seeds, and choice bits encoding the share.

The encryption-based paths already saved their key binding: the ciphertext C_α of the key α for SHE, and the encrypted MAC-key shares under the parties' public keys for LHE. However, if a separately supplied MAC-key share did not match the stored share, the implementation automatically adjusted the ciphertexts to match it. After the patch, these paths use the share stored with the setup and retain the loaded ciphertexts without this adjustment [Kel26a].

We have explicit permission from the maintainer to publish this attack.

Blind authentication check Π_{MACCheck}

Input. The parties hold an authenticated vector $\langle \vec{x} \rangle^\alpha = (\langle x_1 \rangle^\alpha, \ldots, \langle x_n \rangle^\alpha)$ and additive shares $[\alpha]_i$ of the MAC key.

1. Jointly sample public random coefficients $c_1, \ldots, c_n \in \mathbb{F}$ and consume a fresh uniform authenticated mask $\langle r \rangle^\alpha$ from preprocessing.
2. Locally batch and mask the vector:
$$\langle t \rangle^\alpha \leftarrow \sum_{j=1}^{n} c_j \langle x_j \rangle^\alpha + \langle r \rangle^\alpha.$$
3. Open the masked value t, but not its MAC. Each party P_i computes the plain additive check share
$$s_i \leftarrow [m_t]_i - t[\alpha]_i,$$
where $[m_t]_i$ is its MAC share on t.
4. Open $s = s_0 + s_1$ using commit-and-reveal. Accept if $s = 0$; otherwise abort.

Figure 5: The blind authentication check from [SYB+24], showing how to check a vector of authenticated shares without revealing its values.

7 An Aside on Blind Authentication in Shuffling

The dependence on consistent preprocessing MAC keys extends beyond SPDZ circuit evaluation. Several maliciously secure shuffling protocols use blind authentication to check whether secret-shared data and their MACs agree without revealing the data [SYB+24; GZZD24; DNO25; SLD+26; CNOT26].

In this setting, the parties hold a vector of data, with each entry shared using SPDZ authenticated secret sharing. Their goal is to produce authenticated shares of the same entries in a randomly permuted order. Malicious security requires that the data and the overall permutation remain hidden, and that any accepted output contain the same entries as the input. To achieve this last goal, it is useful to check validity after shuffling: the resulting data shares and MAC tags should still form valid authenticated shares. This check is performed without opening the shuffled data, and is a building block in all of the protocols cited above.

However, recall from Section 4.1 that giving an adversary a validity oracle can reveal information about the secret data. A single query with modified shares is enough to test a guess of the secret. The shuffling protocols seemingly provide such an oracle. If their check were just an ideal validity test, the adversary could add 1 to its key share and a guess $\vec{g}$ to its vector of MAC shares after shuffling, and the check would then accept exactly when the shuffled data equaled $\vec{g}$.

Fortunately, the warm-up attack does not immediately break these protocols, because their concrete blind authentication checks use fresh authenticated masks whose authentication depends on preprocessing. Figure 5 presents the batch authentication check from [SYB+24]; its authenticated mask is the $\langle r \rangle^\alpha$ consumed from preprocessing in Step 1. This puts us in the same situation as the masked opening in SPDZ multiplication (Section 4.2). When the mask and the shuffled data are authenticated under the same key, changing the key and correcting the data's MAC for a guess does not also correct the mask's MAC. The modified check therefore does not become an equality test for the guess.

Off-key preprocessing creates the same failure mode as in multiplication. If the mask r in Figure 5 is authenticated under $\alpha + 1$ while the data $\vec{x}$ remains authenticated under α,

the adversary's changes above give

$$s = \sum_{j=1}^{n} c_j(g_j - x_j).$$

The unmasked opening of s therefore reveals the linear combination $\sum_{j=1}^{n} c_j x_j$ to the adversary, by the same argument as in Section 4.4.

Even if this final opening is replaced by μs for a fresh, secret, uniform $\mu \in \mathbb{F}$, as in the masked variant of Figure 2, the check still tests the guess, giving the selective-failure attack of Section 4.3. As in SPDZ, this attack does not contradict the protocols' security guarantees, which require the additive mask r and the data to be authenticated under the same MAC key.

Research Process and AI Use

The Off-Key investigation grew out of our work on secret-shared shuffling protocols. We realized that abstracting blind authentication as an ideal functionality that simply evaluates the MAC-validity equation on the parties' supplied value, MAC, and key shares can itself enable selective-failure attacks, as illustrated in Section 4.1. This observation led us to investigate the security boundary of concrete authentication checks: which assumptions make them secure, and whether implementations preserve those assumptions. This investigation led to the results presented here.

Guided by a specific description of the attack ideas and potential places to look, Anthropic's Claude identified the MP-SPDZ vulnerabilities and implemented the proof of concept. It also implemented the SCALE-MAMBA and FANNG-MPC proofs of concept, but the specific vulnerability was identified by a human.

AI tools, including Claude and Codex, were also used extensively to assist in drafting and revising the manuscript. Automated review agents, primarily Codex and Copilot, provided additional feedback on drafts.

The authors reviewed and heavily revised all AI-generated material. The authors take full responsibility for the paper's claims, code, and text.

Acknowledgements

We thank Xiangfu Song and the other authors of [SYB+24] for noting that SPDZ multiplication can serve as a blind authentication check under inconsistent MAC keys. We also thank Ian Chang and Alex Ozdemir for early discussions of this work.

References

[AAG+25] Najwa Aaraj, Abdelrahaman Aly, Tim Güneysu, Chiara Marcolla, Johannes Mono, Rogerio Paludo, Iván Santos-González, Mireia Scholz, Eduardo Soria-Vazquez, Victor Sucasas, and Ajith Suresh. FANNG-MPC: Framework for Artificial Neural Networks and Generic MPC. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):1–36, 2025. DOI: 10.46586/TCHES.V2025.I1.1-36. URL: https://doi.org/10.46586/tches.v2025.i1.1-36.

[ACC+21] Abdelrahaman Aly, Kelong Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P. Smart, Titouan Tanguy, and Tim Wood. SCALE–MAMBA v1.14: Documentation. Software Documentation, 2021. URL: https://nigelsmart.github.io/SCALE/Documentation-SCALE.pdf.

[AG21] Apple and Google. Exposure Notification Privacy-preserving Analytics (ENPA) White Paper. White Paper, Apple and Google, April 2021. URL: https://covid19.apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

[AOR+19] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE. In Michael Brenner, Tancrède Lepoint, and Kurt Rohloff, editors, *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, pages 33–44. ACM, 2019. DOI: 10.1145/3338469.3358943. URL: https://doi.org/10.1145/3338469.3358943.

[AS22] Damiano Abram and Peter Scholl. Low-Communication Multiparty Triple Generation for SPDZ from Ring-LPN. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part I*, volume 13177 of *Lecture Notes in Computer Science*, pages 221–251. Springer, 2022. DOI: 10.1007/978-3-030-97121-2_9. URL: https://doi.org/10.1007/978-3-030-97121-2_9.

[BCS19] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in Overdrive: A More Efficient ZKPoK for SPDZ. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, volume 11959 of *Lecture Notes in Computer Science*, pages 274–302. Springer, 2019. DOI: 10.1007/978-3-030-38471-5_12. URL: https://doi.org/10.1007/978-3-030-38471-5_12.

[Bea91] Donald Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991. DOI: 10.1007/3-540-46766-1_34. URL: https://doi.org/10.1007/3-540-46766-1_34.

[BKK+16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. *Proc. Priv. Enhancing Technol.*, 2016(3):117–135, 2016. DOI: 10.1515/POPETS-2016-0019. URL: https://doi.org/10.1515/popets-2016-0019.

[BN24] Mako Bates and Joseph P. Near. DT-SIM: Property-Based Testing for MPC Security. *CoRR*, abs/2403.04991, 2024. DOI: 10.48550/ARXIV.2403.04991. arXiv: 2403.04991. URL: https://doi.org/10.48550/arXiv.2403.04991.

[BP26] Luís T. A. N. Brandão and René Peralta. NIST First Call for Multi-Party Threshold Schemes. NIST Internal Report 8214C, National Institute of Standards and Technology, January 2026. DOI: 10.6028/NIST.IR.8214C. URL: https://doi.org/10.6028/NIST.IR.8214C.

[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 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, May 10-14, 2026, Proceedings, Part III*, volume 16543

of *Lecture Notes in Computer Science*, pages 457–487. Springer, 2026. DOI: 10.1007/978-3-032-25324-8_16. URL: https://doi.org/10.1007/978-3-032-25324-8_16.

[Cle86] Richard Cleve. Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract). In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369. ACM, 1986. DOI: 10.1145/12130.12168. URL: https://doi.org/10.1145/12130.12168.

[CNOT26] Ian Chang, Sela Navot, Alex Ozdemir, and Nirvan Tyagi. FLOSS: Fast Linear Online Secret-Shared Shuffling. In *35th USENIX Security Symposium (USENIX Security 26)*, pages 2963–2982, Baltimore, MD. USENIX Association, August 2026. ISBN: 978-1-939133-58-8. URL: https://www.usenix.org/conference/usenixsecurity26/presentation/chang-ian.

[DGN+17] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient Actively Secure Two-Party Computation from Oblivious Linear Function Evaluation. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2263–2276. ACM, 2017. DOI: 10.1145/3133956.3134024. URL: https://doi.org/10.1145/3133956.3134024.

[DI05] Ivan Damgård and Yuval Ishai. Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2005. DOI: 10.1007/11535218_23. URL: https://doi.org/10.1007/11535218_23.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013. DOI: 10.1007/978-3-642-40203-6_1. URL: https://doi.org/10.1007/978-3-642-40203-6_1.

[DNO25] Samuel Dittmer, Rohit Nema, and Rafail Ostrovsky. Linear Secret-shared Shuffle with Malicious Security. *IACR Cryptol. ePrint Arch.*, 2025:2137, 2025. DOI: 10.1007/978-3-032-35418-1_10. URL: https://eprint.iacr.org/2025/2137.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012. DOI: 10.1007/978-3-642-32009-5_38. URL: https://doi.org/10.1007/978-3-642-32009-5_38.

[EB22] Saba Eskandarian and Dan Boneh. Clarion: Anonymous Communication from Multiparty Shuffling Protocols. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. DOI: 10.14722/ndss.2022.24141. URL: https://www.ndss-symposium.org/ndss-paper/auto-draft-243/.

[ETZS26] Hendrik Eerikson, Kert Tali, Anton Zakatov, and Kristjan-Hans Sillmann. JOCONDE D5.1: Demonstration Prototype, Test Cases and Test Results. Technical Report D-16-1232, Cybernetica, May 2026. URL: https://cros.ec.europa.eu/group/6/files/3114/download/2239. Version 1.0. Project JOCONDE, Eurostat and Cybernetica.

[FAN25] FANNG-MPC developers. FANNG-MPC Source Revision d2aa4ad2. Software Source Snapshot, 2025. URL: https://github.com/Crypto-TII/FANNG-MPC/commit/d2aa4ad268a2d27f9b9d2611290035dd2a7b32f6.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure Multi-Party Computation without Agreement. *J. Cryptol.*, 18(3):247–287, 2005. DOI: 10.1007/S00145-005-0319-Z. URL: https://doi.org/10.1007/s00145-005-0319-z.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987. DOI: 10.1145/28395.28420. URL: https://doi.org/10.1145/28395.28420.

[GPP+26] Tim Geoghegan, Christopher Patton, Brandon Pitman, Eric Rescorla, and Christopher A. Wood. Distributed Aggregation Protocol for Privacy Preserving Measurement. Internet-Draft draft-ietf-ppm-dap-19, Internet Engineering Task Force, July 2026. 122 pages. URL: https://datatracker.ietf.org/doc/draft-ietf-ppm-dap/19/. Work in Progress.

[GZZD24] Jiacheng Gao, Yuan Zhang, Sheng Zhong, and Changyu Dong. Free Linear Online Phase for Secure Multiparty Shuffle. Cryptology ePrint Archive, Paper 2024/1936, 2024. URL: https://eprint.iacr.org/2024/1936. Revised September 10, 2026.

[IKK+11] Yuval Ishai, Jonathan Katz, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. On Achieving the "Best of Both Worlds" in Secure Multiparty Computation. *SIAM J. Comput.*, 40(1):122–141, 2011. DOI: 10.1137/100783224. URL: https://doi.org/10.1137/100783224.

[ISO23] ISO/IEC. ISO/IEC 4922-1:2023: Information Security — Secure Multiparty Computation — Part 1: General. International Standard, first edition, July 2023. URL: https://www.iso.org/standard/80508.html.

[ISO24] ISO/IEC. ISO/IEC 4922-2:2024: Information Security — Secure Multiparty Computation — Part 2: Mechanisms Based on Secret Sharing. International Standard, first edition, March 2024. URL: https://www.iso.org/standard/80514.html.

[ITU21] ITU-T. Technical Guidelines for Secure Multi-Party Computation. Recommendation X.1770, International Telecommunication Union, October 2021. URL: https://www.itu.int/rec/T-REC-X.1770-202110-I/en.

[Kel20] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1575–1590. ACM, 2020. DOI: 10.1145/3372297.3417872. URL: https://doi.org/10.1145/3372297.3417872.

[Kel26a] Marcel Keller. Fixed security bug: malicious parties are not committed to MAC key across threads and executions. Software Commit, 2026. URL: https://github.com/data61/MP-SPDZ/commit/982e91e7a3784c5b56c2db30813767a485e6741f.

[Kel26b] Marcel Keller. MP-SPDZ Source Revision 58d2739a. Software Source Snapshot, 2026. URL: https://github.com/data61/MP-SPDZ/commit/58d2739ac251262c0bd6034f7fc7d0666a6b1e2c.

[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, *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, pages 2491–2508. IEEE, 2025. DOI: 10.1109/SP61157.2025.00176. URL: https://doi.org/10.1109/SP61157.2025.00176.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842. ACM, 2016. DOI: 10.1145/2976749.2978357. URL: https://doi.org/10.1145/2976749.2978357.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ Great Again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189. Springer, 2018. DOI: 10.1007/978-3-319-78372-7_6. URL: https://doi.org/10.1007/978-3-319-78372-7_6.

[Lau21] Peeter Laud. Linear-Time Oblivious Permutations for SPDZ. In Mauro Conti, Marc Stevens, and Stephan Krenn, editors, *Cryptology and Network Security - 20th International Conference, CANS 2021, Vienna, Austria, December 13-15, 2021, Proceedings*, volume 13099 of *Lecture Notes in Computer Science*, pages 245–252. Springer, 2021. DOI: 10.1007/978-3-030-92548-2_13. URL: https://doi.org/10.1007/978-3-030-92548-2_13.

[LDS+25] Guopeng Lin, Xiaoning Du, Lushan Song, Weili Han, Jin Tan, Junming Ma, Wenjing Fang, and Lei Wang. Is MPC Secure? Leveraging Neural Network Classifiers to Detect Data Leakage Vulnerabilities in MPC Implementations. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, pages 2809–2826. IEEE, 2025. DOI: 10.1109/SP61157.2025.00181. URL: https://doi.org/10.1109/SP61157.2025.00181.

[Lin23] Yehuda Lindell. Digital Asset Management with MPC (Whitepaper). Coinbase Blog, June 2023. URL: https://www.coinbase.com/blog/digital-asset-management-with-mpc-whitepaper.

[MP-26a] MP-SPDZ developers. MP-SPDZ Security Fixes. Software Documentation, 2026. URL: https://mp-spdz.readthedocs.io/en/latest/security-fixes.html. Accessed 2026-10-03.

[MP-26b] MP-SPDZ developers. MP-SPDZ v0.4.3. Software Release, 2026. URL: https://github.com/data61/MP-SPDZ/releases/tag/v0.4.3.

[NS23] Raine Nieminen and Thomas Schneider. Breaking and Fixing Garbled Circuits When a Gate has Duplicate Input Wires. *J. Cryptol.*, 36(4):34, 2023. DOI: 10.1007/S00145-023-09472-4. URL: https://doi.org/10.1007/s00145-023-09472-4.

[RS22] Rahul Rachuri and Peter Scholl. Le Mans: Dynamic and Fluid MPC for Dishonest Majority. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 719–749. Springer, 2022. DOI: 10.1007/978-3-031-15802-5_25. URL: https://doi.org/10.1007/978-3-031-15802-5_25.

[SCA22a] SCALE-MAMBA developers. SCALE-MAMBA Software. Software Project Website, 2022. URL: https://nigelsmart.github.io/SCALE/. Project history and June 2022 maintenance notice. Accessed 2026-10-03.

[SCA22b] SCALE-MAMBA developers. SCALE-MAMBA Source Revision c111516e. Software Source Snapshot, 2022. URL: https://github.com/KULeuven-COSIC/SCALE-MAMBA/commit/c111516e3ebc1efd12a2bd47dd2122b160e13d1e.

[SLD+26] Xiangfu Song, Xiaojian Liang, Ye Dong, Jianli Bai, Pu Duan, Changyu Dong, Tianwei Zhang, and Ee-Chien Chang. Secret-Shared Shuffle from Authenticated Correlations. In Shi Bai and Edoardo Persichetti, editors, *Public-Key Cryptography - PKC 2026 - 29th IACR International Conference on Practice and Theory of Public-Key Cryptography, West Palm Beach, FL, USA, May 25-28, 2026, Proceedings, Part II*, volume 16552 of *Lecture Notes in Computer Science*, pages 321–354. Springer, 2026. DOI: 10.1007/978-3-032-26734-4_11. URL: https://doi.org/10.1007/978-3-032-26734-4_11.

[SYB+24] Xiangfu Song, Dong Yin, Jianli Bai, Changyu Dong, and Ee-Chien Chang. Secret-Shared Shuffle with Malicious Security. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. DOI: 10.14722/ndss.2024.24021. URL: https://www.ndss-symposium.org/ndss-paper/secret-shared-shuffle-with-malicious-security/.

[XZ24] Zheng Xu and Yanxiang Zhang. Advances in Private Training for Production On-Device Language Models. Google Research Blog, February 2024. URL: https://research.google/blog/advances-in-private-training-for-production-on-device-language-models/.