IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–22. DOI:XXXXXXXX

Fault Injection Attacks Against zkSTARKs

Alexander Dalton[1], Daniel Page[2] and Markus Schofnegger[3]

[1] Independent
[2] University of Bristol, Bristol, UK
[3] [[alloc] init], San Francisco, United States

Abstract. Fault injection attacks on zero-knowledge (ZK) constructions have received little attention in the wider literature. Recent work has demonstrated single-fault key recovery attacks against a ZK signature scheme. In this paper, we detail candidate fault injection attacks against zero-knowledge scalable transparent argument of knowledge (zkSTARK) provers, designed to violate ZK capabilities. zkSTARK proving systems are complex, and implementations differ substantially in their implementation specifics. We propose a corresponding variety of fault injection attacks against different popular algorithmic primitives. To the best of our knowledge, this marks the first exploration of the fault injection surface of zkSTARKs, and of the wider class of general-purpose ZK proving systems.

Keywords: Fault Injection · Zero Knowledge Proofs · zkSTARKs

1 Introduction

Zero-knowledge proof protocols (ZKPs) involve two parties, namely a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$, and allow $\mathcal{P}$ to convince $\mathcal{V}$ that a statement is valid without revealing any information beyond validity of the statement. A given ZKP can be classified as being either interactive or non-interactive, i.e., a non-interactive zero-knowledge (NIZK) protocol, with the latter often constructed from the former via application of the Fiat–Shamir transformation [FS86]. Whereas an interactive ZKP allows multiple rounds of communication between $\mathcal{P}$ and $\mathcal{V}$, a NIZK protocol implies a single round of communication, in which $\mathcal{P}$ convinces $\mathcal{V}$ that the statement is valid using a single communicated message.

One can view the step from a ZK to a NIZK protocol as optimising with respect to interactivity to enable practicality, with further sub-classes of ZKP then optimising for other properties. For example, a ZKP is termed a zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) if it affords *non-interactivity*, and *succinctness*, the latter property requiring proofs to be small. A ZKP is termed a zkSTARK if it additionally affords *transparency*, avoiding the trusted initialisation phase required by a zkSNARK, and *scalability*, allowing proof generation and verification to remain efficient even for complex statements. Note that the term *argument* rather than *proof* reflects the assumption of a computationally bounded $\mathcal{P}$. Associated frameworks for their construction are instantiated using 1) a mechanism to represent the statement such as arithmetisation for circuit-like statements, and 2) a proof system, potentially translated (or compiled) into the resulting protocol. For example, use of a hash- rather than DLP-based proof system, as typical in a zkSTARK and a zkSNARK respectively, can afford plausible post-quantum security.

The Threat Landscape for ZK-Based Constructions. Note that the ZK property underlying ZKPs can be intuitively aligned with many use cases, but particularly those which involve privacy preservation. For example, an edge device may use a ZK proof to authorise

Licensed under Creative Commons License CC-BY 4.0.

a private cryptocurrency transfer [BSCG+14]. This and other examples can therefore pitch ZKPs within contexts for which the threat of implementation attacks is pertinent. For example, one can imagine an attacker targeting an implementation executing on some target device using passive, i.e., side-channel, and/or active, i.e., fault injection, techniques.

The framing of a ZKP as a complex, composite protocol rather than a primitive hints at various challenges with respect to mitigation of such attacks. First, the attack *surface* can extend beyond a single target device into third-party devices (and/or wider systems). Examples include those required to support trusted initialisation, and/or computational off-load (as may be required for proof generation from a constrained edge device). Note that the relationship between such devices can be non-trivial in terms of function, but also trust. Second, the attack *goal* can extend beyond key recovery. The attacker may target private witnesses, secret keys, or other information protected by the zero-knowledge property. Third, the number and complexity of components which comprise a given ZKP often lead to a high-level implementation being compiled into a lower-level, deployed executable. In common with non-ZKP settings (see, e.g., [SCA18, SLP+25]), this fact leads to the potential for and therefore also the need to mitigate compiler-induced vulnerabilities.

Scope and Contributions. We suggest that, in part because of the relative immaturity of ZKPs in relevant contexts, there is a shortage of work on and thus understanding of implementation attacks and associated countermeasures. The most relevant related work is that of Mondal et al. [MAKK24], who focus on ZK-based post-quantum digital signature schemes. By injecting a single fault, they were able to invalidate the ZK property and allow $\mathcal{V}$ to learn information about secret data involved. Other, less directly relevant related work includes successful application of side-channel attacks. Examples include extraction of secret data in fast Reed–Solomon interactive oracle proof of proximity (FRI)-based ZK membership proofs [MRS25] and in ZK-based cryptocurrencies [TBP20].

The rapid evolution and deployment of ZKPs as a component of numerous, diverse use cases implies that extension of work in this field is of some importance. As such, in this work we study fault injection as applied to zkSTARKs specifically. In more concrete terms, we make the following contributions:

1. In Section 3.1 we give conditions under which revealing witness-dependent polynomial evaluations permits distinguishing between private transcripts.
2. Following this intermediate result we develop and evaluate various attacks, differing with respect to the target operation or fault model. These include:
   - An attack against point-wise polynomial masking based on an instruction-skip fault model.
   - Two attacks (varying by Cooley–Tukey variant) against interleaved transcript paddings via iterative Cooley–Tukey number theoretic transform (NTT) based on a single-bit-upset fault model.
   - Two attacks (again, varying by Cooley–Tukey variant) against concatenated transcript paddings via recursive Cooley–Tukey NTT based on a single-bit-upset and instruction-skip fault model.
3. We validate each attack experimentally via an instruction-level RISC-V simulation.
4. We further develop and discuss various countermeasures, with various trade-offs in terms of cost and capability.

Note that all associated material (e.g., source code relating to all software implementations) is available under an open source license.[1]

[1] https://github.com/AlexDalt/spike-fi

2 Preliminaries

2.1 Notation

Throughout the paper, we write $\mathcal{P}$ and $\mathcal{V}$ for the two parties of a proof system, namely the prover and the verifier. We work over a finite field $\mathbb{F}$, and we use the lowercase letters f, p, and c for polynomials over $\mathbb{F}$. The statement that $\mathcal{P}$ wants to prove concerns a private execution transcript A, which is the sequence of field elements produced by running the computation of interest, and whose length we denote by $n = |A|$. Two evaluation domains appear throughout our description. The first is a small transcript domain $G \subset \mathbb{F}$ on which the transcript lives, and the second is a strictly larger extension domain $L \subseteq \mathbb{F}$ with $|L| = \rho^{-1}|A|$, where $\rho^{-1} > 1$ is a small blow-up factor. Finally, commitments are realised as Merkle trees, and we write M_f and M_p for the trees committing to the transcript polynomial and to the composition polynomial, respectively. Requesting the value stored at a given leaf together with its authentication path is what we call opening the commitment. All logarithms are taken to base 2. λ denotes the security parameter.

2.2 STARKs

In this section we describe the components of a scalable transparent argument of knowledge (STARK) required to understand our attacks. We omit components not pertinent to our attacks, namely the internals of the low-degree test. We refer the reader to [BBHR18b] for the full construction. We focus on univariate STARKs, where the data of the prover is encoded into univariate polynomials. Our attacks may also apply in the multivariate setting, but we did not test this case.

At a high level, a STARK allows $\mathcal{P}$ to convince $\mathcal{V}$ that a given computation was carried out correctly, without $\mathcal{V}$ having to recompute it. In general, the construction can be split into the five steps described in the following, which we also summarise in Fig. 1. Our attacks target transcript interpolation, masking, and low-degree-extension computation (also referred to as *batch evaluation generation* in this work), before the transcript commitment is released.

Arithmetisation. In the first step, the computation is written as an execution transcript, which is a table of field elements. Each column of this table follows one register over time, and each row corresponds to one step of the computation. Reading a column from top to bottom gives the private transcript A. The correctness of the computation is then expressed by a set of public algebraic constraints that every honest transcript must satisfy. Here, boundary constraints fix certain cells to fixed values (e.g., the inputs and outputs of the computation), while transition constraints relate each row to the next one and in this way encode the step function of the computation. A transcript is valid if and only if all of these constraints vanish on every required row.

Transcript Polynomials. The transcript domain G is chosen as a multiplicative subgroup of $\mathbb{F}$ of size $|A|$, generated by an element g, so that $G = \{g^0, g^1, \ldots, g^{|A|-1}\}$. Each column of the transcript is then interpolated into the unique polynomial $f \in \mathbb{F}[X]$ of degree smaller than $|A|$ that agrees with the column on G, i.e., $A = \{f(g^0), f(g^1), \ldots, f(g^{n-1})\}$.

Low-Degree Extension and Commitment. In order to make the polynomial testable, $\mathcal{P}$ evaluates the polynomial f on the larger extension domain L. The resulting vector of evaluations forms a Reed–Solomon codeword, and its redundancy is what the subsequent proximity test relies on. The Reed–Solomon expansion will be important later when we attack interleaved padding, as the expansion affects the algorithmic stage we attack. These evaluations are stored in the leaves of a Merkle tree M_f, whose root $\mathcal{P}$ sends to $\mathcal{V}$ as a

binding commitment. From this point on, $\mathcal{V}$ can ask for any leaf to be opened, and the corresponding authentication path prevents $\mathcal{P}$ from changing a value afterwards.

Composition Polynomial. Using random weights provided by $\mathcal{V}$, the individual constraints are then combined into a single composition polynomial p. This polynomial is constructed such that it is of low degree if the transcript satisfies all constraints (and, with high probability, only then), which is in turn equivalent to the constraint polynomial c vanishing on the entire domain G. The prover evaluates p over L and commits to it in a second Merkle tree M_p. In this way, checking that the computation was carried out correctly reduces to checking that p is of low degree. We note that only M_f is relevant for our attacks, and we mention M_p only for completeness.

Low-Degree Test and Query Phase. Finally, $\mathcal{P}$ and $\mathcal{V}$ run the FRI protocol [BBHR18a], an interactive oracle proof which shows that the committed evaluations are close to a low-degree codeword. The protocol proceeds in rounds, and each round halves the degree that is being tested. It ends with a query phase, in which $\mathcal{V}$ chooses a number of random positions in L, and $\mathcal{P}$ opens M_f, M_p, and the intermediate FRI commitments at these positions, together with the corresponding authentication paths. The number of queried positions is determined by the security parameter. If all openings are consistent and satisfy the FRI relation, then p is of low degree except with a small and fixed soundness error, and hence the constraints hold and the proof is accepted. As we will see, it is exactly these openings of M_f that our attacks try to steer towards values carrying evaluations of an unmasked f.

Non-Interactivity. It is common to apply the Fiat–Shamir transform [FS86] in order to turn the interactive parts of the protocol into a non-interactive proof. In this case, all randomness of the verifier (i.e., the constraint weights, the FRI folding challenges, and the query positions) is instead derived by hashing the transcript of messages exchanged so far. All our attacks target the non-interactive setting.

2.3 Zero Knowledge in STARKs

The STARK construction given above only allows $\mathcal{P}$ to convince $\mathcal{V}$ that a computation was executed correctly, and provides no zero-knowledge functionality. We note that in some cases, FRI and the entire proving stack are used without actually activating the zero-knowledge feature. This is the case for many popular zero-knowledge virtual machines (zkVMs), where the main properties exploited are succinctness and computational integrity rather than enforcing privacy [Suc].

However, there are use cases (e.g., [Azt21]) where ZK is indeed required. In these cases the verifier should learn no witness information beyond what is implied by the public statement. For our attacks, we have to consider a scenario where ZK is enabled, otherwise they would be meaningless.

There are three primary mechanisms used "in the wild" to ensure privacy, namely polynomial masking, random transcript padding, and using an outer ZK-enabled proof system. We exclude the third mechanism, as any ZK properties are delegated to a separate proof system are beyond the scope of this work.

Polynomial masking involves replacing each polynomial (e.g., f) in the protocol with a random polynomial (e.g., f') that agrees with the original over the transcript domain ($\forall g \in G, f(g) = f'(g)$). Concretely, this is usually achieved by setting $f'(X) = f(X) + Z_G(X) \cdot r(X)$, where Z_G is a polynomial that vanishes over G (i.e., $\forall g \in G, Z_G(g) = 0$) and r is a random low-degree polynomial. Common convention sets $Z_G(X) = X^n - 1$.

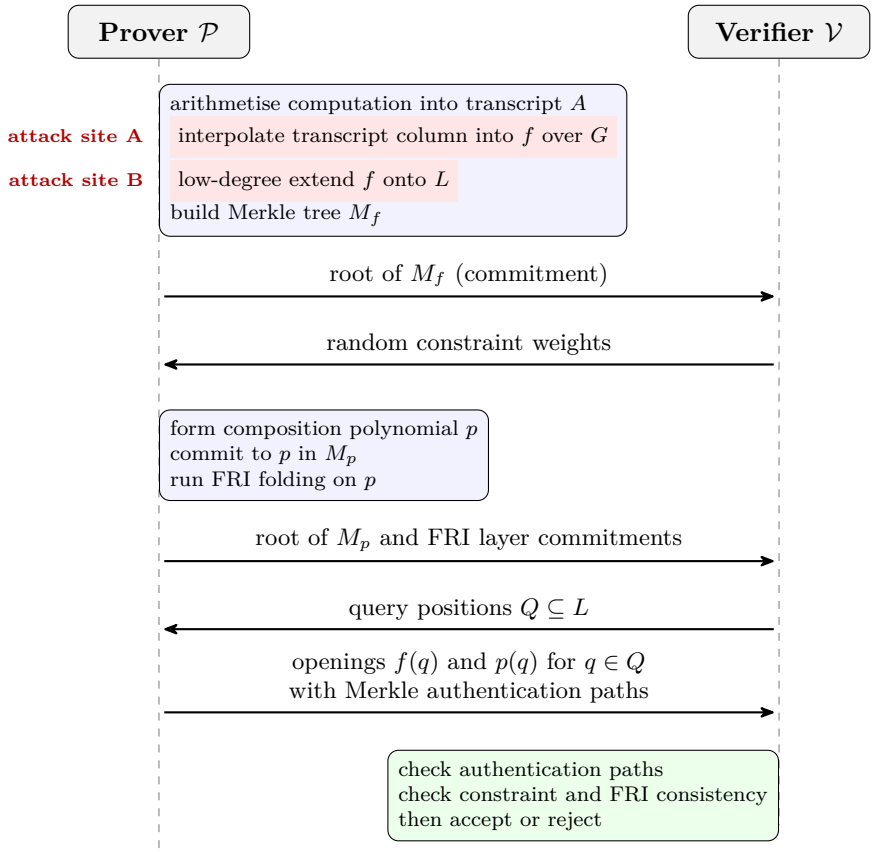

Figure 1: High-level message flow of a univariate STARK, with $\mathcal{P}$ on the left and $\mathcal{V}$ on the right. The boxes denote local computations, while the arrows denote exchanged messages. When the Fiat–Shamir transform is applied, $\mathcal{P}$ computes the messages from $\mathcal{V}$ as hashes of the previous transcript instead of receiving them. Our attacks target the interpolation, masking, and low-degree extension steps in the top box, and they do so by influencing which evaluations of f are eventually opened to $\mathcal{V}$. Our attacks against interpolation occur at the labelled attack site A, and our attacks against polynomial masking and batch evaluation generation occur at attack site B.

Random transcript padding involves adding random values to an existing witness column in the transcript, in order to mask the potentially private witness values by disassociating the interpolated protocol polynomial from the private transcript. The random values are either appended to the column (*concatenated transcript padding*) or interleaved with its entries (*interleaved transcript padding*). We refer to [HK24] for details on this approach and an overview of different techniques. An additional step consists of masking the Merkle tree leaves with random paddings, in order to prevent information leakage when opening values in the course of the protocol. This technique is described in more detail in [BCS16], although this approach is not widespread in existing zkSTARK implementations.

2.4 ZK Proof Attack Surface

When considering proving system security, a prover's host system is an obvious target. However, the attack surface for proving system vulnerabilities goes beyond compromised

provers, and can be quite subtle. Many ZK systems have a complicated relationship between different components and third parties with different degrees of trust. A somewhat common complication involves delegating proof generation to a powerful trusted third party [Mid26]. Constructing zero-knowledge proofs is computationally expensive, and as such, some projects suggest offloading proof generation to a trusted third-party service. When proof generation is offloaded to a third party the user is exposed to the risk that even if a third party has honest intentions, they may be unknowingly compromised and subject to adversarial attack.

For example, the trusted third party may well be running a proof-generation-as-a-service, and running multiple workloads on shared hardware. This parallelisation risks honestly run proof generation interacting unpredictably with external malicious workloads. Trusted third parties, especially those servicing many users, make for a tempting single point of failure for an adversarial actor and increase the attack surface of an individual interacting with a ZK proof system.

2.5 Threat Model

The attacks presented in this work require the adversary to have one or both of two possible capabilities: the ability to induce *instruction skips*, or *single-bit upsets*. The former, in particular, requires a high degree of timing precision. Both are widely understood to be plausible fault models [KSV13].

Instruction Skip. An instruction skip suppresses the execution of a selected instruction. Instruction skips can be manufactured with clock glitching and are known to be reproducible with high probability.

Single-Bit Upset. Single-bit upsets are faults that induce a single bit flip in a target register. These types of faults are often induced with lasers targeting physical locations on hardware.

2.6 Existing Implementations

Existing zkSTARK implementations differ in their algorithmic primitives and ZK mechanisms. In this section we survey existing implementations and detail the specific algorithms and ZK mechanisms we will later attack. Most of the attacks in this work target one key primitive, namely the Cooley–Tukey NTT algorithm [CT65] for polynomial interpolation and the generation of batch polynomial evaluations. We will construct attacks against each ZK mechanism. Specifically, we target four prevalent Cooley–Tukey variants observed in the survey: decimation-in-time (DIT)/decimation-in-frequency (DIF) recursive, and DIT/DIF iterative.

Table 1 details the Cooley–Tukey variant and ZK mechanism(s) used in each surveyed implementation. In this work we do not attack these implementations directly, instead reimplementing the core components and attacking these in a single unified environment.

To illuminate a specific example in Table 1, RISC Zero [RISa] implements a general-purpose zkVM that allows arbitrary computations to be executed privately while producing a publicly verifiable proof of correct execution. Unlike a number of STARK implementations which provide only computational integrity, RISC Zero implements an explicit ZK mechanism by appending cryptographically random values to each execution transcript before polynomial interpolation, a technique referred to in this work as *concatenated transcript padding*.

In Table 2 we distil Table 1 into a matrix detailing which Cooley–Tukey variant is used in conjunction with which ZK mechanism. This matrix guides the attacks explored in this work, ensuring we focus on pairings witnessed in the real world. Related attacks against

Table 1: Zero-knowledge mechanisms and polynomial-transform algorithms used in public STARK implementations.

Implementation	Zero-Knowledge Mechanism	Cooley–Tukey Variant
Stone Prover [Stab]	No zero knowledge	
Winterfell [Fac]	No zero knowledge	
Distaff [Guia]	No zero knowledge	
ethSTARK [Staa]	No zero knowledge	
Stwo [Stac]	No zero knowledge	
Plonky2 [Polb]	optional witness blinding and Merkle-leaf salting	Iterative DIT
Plonky3 [Plo]	Polynomial masking separately evaluated added point-wise; interleaved transcript padding	Iterative DIF/DIT
RISC Zero [RISa]	Concatenated transcript padding	Recursive DIT/DIF
zkSync Boojum [Mata]	No zero knowledge	
Polygon zkEVM [Pola]	Secondary recursive proof	
Lambdaworks [Lam]	No zero knowledge	
SP1 [Suc]	Secondary recursive proof	
OpenZKP [0xP]	No zero knowledge	
Hodor [Matb]	No zero knowledge	
Valida [Val]	No zero knowledge	
libSTARK [Eli]	Polynomial masking with evaluations taken directly from large masked polynomial	None: additive/subspace FFT
genSTARK [Guib]	No zero knowledge	
Triton VM [Tri]	Polynomial masking separately evaluated added point-wise	Iterative DIT

unexplored mechanism/Cooley–Tukey variant pairs can be easily derived given the attacks specified in this work. We leave additive/subspace FFT attacks to future work.

Given Table 2, we select the following attack targets: polynomial masking (evaluated separately with point-wise addition), interleaved transcript padding with both iterative DIT and iterative DIF Cooley–Tukey, and concatenated transcript padding with both recursive DIT and recursive DIF Cooley–Tukey.

2.7 Algorithm Definitions

The majority of the attacks presented in this work aim to subvert one of the ZK techniques presented in the previous section by targeting the NTT algorithm. The Cooley–Tukey NTT algorithm is used in many of the surveyed implementations to interpolate the execution transcript into a private polynomial f, and to generate batch evaluations of the private transcript polynomial f.

It is a common convention to make use of the Cooley–Tukey NTT algorithm as a trick to calculate batch polynomial evaluations. This trick works as a STARK implementation does not care what the extended evaluation domain L actually contains, and the output of an NTT is structurally identical to repeated evaluations of a polynomial over a large domain $W = \{\omega^i\}$ of roots of unity. The Cooley–Tukey algorithm is one of the most popular NTT algorithms, and as such, many implementations in Table 1 use a variant of Cooley–Tukey for batch polynomial evaluations. The Cooley–Tukey algorithm in general operates via a divide-and-conquer approach. It reorganises a single NTT into two smaller NTTs that must later be mixed. The DIT or DIF distinction refers to the order in which the smaller NTTs are executed, with the former performing the smallest NTT first and progressively larger NTTs after, and the latter performing the largest NTT first and

Table 2: Observed combinations of zero-knowledge mechanisms and enabling transform algorithms in public STARK implementations. A ✓ denotes a combination observed in at least one surveyed implementation, while × denotes a combination not observed.

Zero-Knowledge Mechanism	**Iterative DIT**	**Iterative DIF**	**Recursive DIT**	**Recursive DIF**	**Additive/Subspace FFT**
Polynomial masking: separate evaluation and point-wise addition	✓	✓	×	×	×
Polynomial masking: direct evaluation of masked polynomial	×	×	×	×	✓
Interleaved transcript padding	✓	✓	×	×	×
Concatenated transcript padding	×	×	✓	✓	×

progressively smaller NTTs after. The algorithm can run either recursively or iteratively in both the DIT and DIF cases. The algorithms are specified in their entirety for completeness. We refer to the sub-NTTs as *mixing blocks*, with 1-sized mixing blocks performing an NTT on neighbouring elements, and n-sized mixing blocks performing an NTT across elements spaced n elements apart.

2.7.1 DIF Iterative Cooley–Tukey NTT

Algorithm 1 specifies the iterative DIF transform, which performs larger butterfly stages before smaller ones.

Algorithm 1 Cooley–Tukey NTT (DIF, in-place, iterative)

procedure NTT-DIF(a, n, p, ω)
  for $s = \log_2 n$ down to 1 **do**
    $m \leftarrow 2^s$
    $\omega_m \leftarrow \omega^{n/m} \bmod p$
    for $k = 0$ to $n - 1$ by m **do**
      $\omega_{\text{current}} \leftarrow 1$
      for $j = 0$ to $m/2 - 1$ **do**
        $u \leftarrow a[k + j]$
        $v \leftarrow a[k + j + m/2]$
        $a[k + j] \leftarrow (u + v) \bmod p$
        $d \leftarrow (u - v + p) \bmod p$
        $a[k + j + m/2] \leftarrow \omega_{\text{current}} \cdot d \bmod p$
        $\omega_{\text{current}} \leftarrow \omega_{\text{current}} \cdot \omega_m \bmod p$
  Bit-reverse reorder a
  return a

2.7.2 DIT Iterative Cooley–Tukey NTT

In contrast to the DIF iterative Cooley–Tukey NTT algorithm specified above, the DIT variant operates on small sub-NTTs first and large sub-NTTs last. The details of the

algorithm are specified in Algorithm 2.

Algorithm 2 Cooley–Tukey NTT (DIT, in-place, iterative)

procedure NTT-DIT(a, n, p, ω)
  Bit-reverse reorder a
  for $s = 1$ to $\log_2 n$ **do**
    $m \leftarrow 2^s$
    $\omega_m \leftarrow \omega^{n/m} \bmod p$
    for $k = 0$ to $n - 1$ by m **do**
      $\omega_{\text{current}} \leftarrow 1$
      for $j = 0$ to $m/2 - 1$ **do**
        $t \leftarrow \omega_{\text{current}} \cdot a[k + j + m/2] \bmod p$
        $u \leftarrow a[k + j]$
        $a[k + j] \leftarrow (u + t) \bmod p$
        $a[k + j + m/2] \leftarrow (u - t + p) \bmod p$
        $\omega_{\text{current}} \leftarrow \omega_{\text{current}} \cdot \omega_m \bmod p$
  return a

2.7.3 DIF Recursive Cooley–Tukey NTT

The details of the algorithm are specified in Algorithm 3. In comparison to the iterative case, instead of expressing the algorithm as a series of nested loops this variant uses recursion. Our implementation performs this transform in place, i.e., u and v overwrite the lower and upper half of a, and returns the output in bit-reversed order.

Algorithm 3 Cooley–Tukey NTT (DIF, recursive)

procedure NTT-Recursive-DIF(a, n, p, ω)
  if $n = 1$ **then**
    return a
  $\omega_{\text{current}} \leftarrow 1$
  for $j = 0$ to $n/2 - 1$ **do**
    $u[j] \leftarrow (a[j] + a[j + n/2]) \bmod p$
    $v[j] \leftarrow (a[j] - a[j + n/2] + p) \bmod p$
    $v[j] \leftarrow v[j] \cdot \omega_{\text{current}} \bmod p$
    $\omega_{\text{current}} \leftarrow \omega_{\text{current}} \cdot \omega \bmod p$
  $\omega' \leftarrow \omega^2 \bmod p$
  $U \leftarrow$ NTT-Recursive-DIF$(u, n/2, p, \omega')$
  $V \leftarrow$ NTT-Recursive-DIF$(v, n/2, p, \omega')$
  for $j = 0$ to $n/2 - 1$ **do**
    $y[j] \leftarrow U[j]$
    $y[j + n/2] \leftarrow V[j]$
  return y

2.7.4 DIT Recursive Cooley–Tukey NTT

The details of the algorithm are specified in Algorithm 4. As in the iterative case, in contrast to the DIF variant, the DIT algorithm performs butterfly operations between small blocks first and large blocks last.

Algorithm 4 Cooley–Tukey NTT (DIT, recursive)

procedure NTT-RECURSIVE-DIT(a, n, p, ω)
  if $n = 1$ **then**
    return a
  for $j = 0$ to $n/2 - 1$ **do**
    $a_{\text{even}}[j] \leftarrow a[2j]$
    $a_{\text{odd}}[j] \leftarrow a[2j + 1]$
  $\omega' \leftarrow \omega^2 \bmod p$
  $E \leftarrow$ NTT-RECURSIVE-DIT$(a_{\text{even}}, n/2, p, \omega')$
  $O \leftarrow$ NTT-RECURSIVE-DIT$(a_{\text{odd}}, n/2, p, \omega')$
  $\omega_{\text{current}} \leftarrow 1$
  for $j = 0$ to $n/2 - 1$ **do**
    $t \leftarrow \omega_{\text{current}} \cdot O[j] \bmod p$
    $y[j] \leftarrow (E[j] + t) \bmod p$
    $y[j + n/2] \leftarrow (E[j] - t + p) \bmod p$
    $\omega_{\text{current}} \leftarrow \omega_{\text{current}} \cdot \omega \bmod p$
  return y

2.8 Fault Injection via `spike-fi`

`spike-fi` is a fault injection simulation framework based on the `spike` RISC-V ISA simulator [risb]. Simulating the attacks at the instruction level provides a much closer approximation of mounting a real attack than source-level simulation. In particular, it forces us to consider the compiler artefacts. Instead of deciding certain code paths should be "skipped", we have to isolate specific architectural instructions as targets for fault injection. For example, when attempting to skip a loop one must decide whether to attack the entry conditions to the loop, or whether it would suffice to perform an instruction skip on the return edge of the loop, resulting in a partial loop skip in which the first iteration is guaranteed to run. These effects are not captured in a source-level simulation. Unfortunately, much like source-level simulation, `spike-fi` does not simulate fault-induced side-effects, for example faults that missed the target instruction/register and affected other parts of the program, potentially inducing a crash. Nor does `spike-fi` simulate device-specific physical effects. Physical validation of our attacks remains future work.

3 Attacks

In this section we detail our attacks against zkSTARK provers. We start by proving that leaking evaluations of f violates zero knowledge; this is followed by an overall methodology; then the details of each attack in turn, including theoretical basis and experimental results. Specific countermeasures are included after the relevant attack, with general countermeasures described at the end.

3.1 Intermediate Result

Lemma 1. *Discovering even a single evaluation $y_x = f(x)$ for a known value x and transcript polynomial f has a high probability of violating the zero-knowledge property of a zkSTARK.*

Proof. First, consider an adversary $\mathcal{A}$ capable of extracting a randomly sampled evaluation $y_x = f(x)$ for some known x from a public protocol transcript T produced by the faulted prover Prove^* with some non-negligible probability, i.e.,

$$P\big(\mathcal{A}(T) = (y, x), y = f(x)\big) \geq \frac{1}{\mathsf{poly}(\lambda)}.$$

Given $\mathcal{A}$ we can construct $\mathcal{B}$, specified in Fig. 2, which can distinguish two private transcripts $A_0 \neq A_1$ with different underlying secret witness values but the same public constraints. If $\mathcal{A}$ can extract evaluations of f with non-negligible probability, then $\mathcal{B}$ can distinguish between A_0 and A_1 with good probability by generating reference evaluations $y_0 = f_0(x), y_1 = f_1(x)$ and comparing these values with the output of $\mathcal{A}$.

Given the univariate case of the Schwartz–Zippel lemma [Sch80], distinct polynomials of degree d can agree at at most d points. Since $\mathcal{B}$ generates A_0 and A_1 itself, it can independently interpolate the transcripts into f_0 and f_1. Each invocation of $\mathcal{A}$ returns an evaluation y and its preimage x, so the loop in Fig. 2 simply resamples until it lands on an evaluation where $f_0(x) \neq f_1(x)$, which happens with probability greater than $1/2$ for each returned evaluation, as x is a Fiat–Shamir derived query position that is uniform over at least $2n$ points in our attacks, while the nonzero polynomial $f_0 - f_1$ of degree smaller than n has fewer than n roots. At such an evaluation, the extracted value y is decisive, i.e., it can match at most one of f_0 and f_1. Therefore,

$$\left|\frac{1}{2} - P(\mathcal{B}(T_b) = b)\right| \geq \frac{1}{\mathsf{poly}(\lambda)}.$$

Zero knowledge requires that proofs produced from two valid private transcripts for the same public statement be computationally indistinguishable. This follows because a zero-knowledge simulator is independent of the witness, so the proofs of A_0 and A_1 are each indistinguishable from the same simulated proof, and hence from each other. The distinguisher above contradicts this requirement.

Therefore, learning any evaluation y is probabilistically sufficient to break the zero-knowledge property. □

Beyond this intermediate result, leaking any output of a known witness-dependent function that can be used to distinguish between valid witnesses is sufficient to break ZK, for example, directly witnessing a single coefficient of f. The adversary can replay the known transformation against f_0 and f_1 to distinguish based on the learned value. Furthermore, as f has degree smaller than n, each discovered evaluation constrains the possible private transcript, and if n evaluations are discovered, the polynomial f, and thereby the private transcript A, can be completely recovered via Lagrange interpolation.

3.2 Experimental Methodology

The attacks presented in this work have been validated against `spike-fi`, a RISC-V instruction set simulator with fault injection capabilities. All experiments targeted zkSTARK proving over a simple private computation calculating the first n Fibonacci numbers, with o Merkle tree openings, and some probability of successfully inducing a single fault given by p_f. We counted a trial as successful if at least one value satisfying the leakage predicate appeared in the released transcript.

Obviously, given the public constraints of our target program (the Fibonacci pattern, with a known starting value), it is trivial to derive the private transcript, and as such we ignored the ability for an adversary to derive information from the public constraints. Depending on the specific context the adversary could be further empowered in this way, by exploiting some program-specific contextual information and using the public constraints to derive additional private information. This style of extended attack is left to future work.

In all cases we ran two independent sweeps. The first varied n and p_f with $o = 64$, and the second varied o and p_f with $n = 512$. In all cases the probability of successful

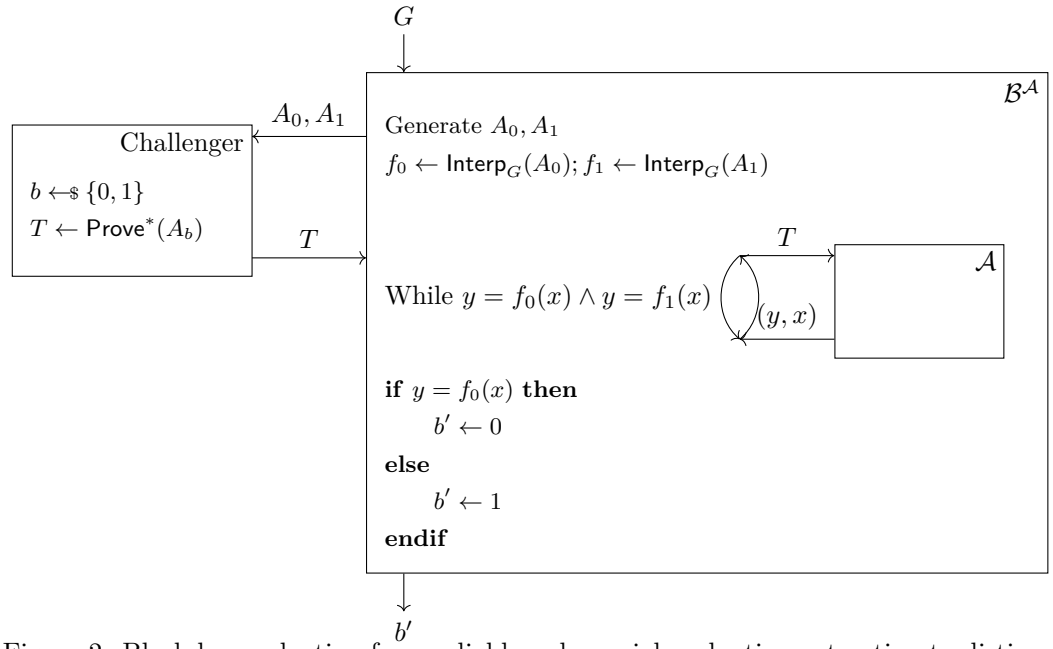

Figure 2: Black-box reduction from reliable polynomial-evaluation extraction to distinguishing private execution transcripts for the same public statement. $\mathcal{B}$ wins if $b' = b$.

fault injection varied between 0.0 and 1.0, with increments of 0.1. Each configuration was repeated across 30 trials.

We assume that the adversary knows sufficient information about a target zkSTARK implementation to select an appropriate attack, can induce faults as stipulated by the fault model, and can observe the public transcript of the operating zkSTARK protocol.

3.3 Attacking Polynomial Masking

As previously discussed, a common technique for adding zero knowledge to a STARK proving system involves replacing the private transcript polynomial f with a random polynomial f' which agrees with f on the transcript domain G. Whenever f is required in the unmasked protocol, in the masked variant f' is instead used, and by disassociating the protocol polynomial from the private transcript, the protocol leaks no information about the underlying transcript other than the correctness of the computation. The attack defined in this section performs a single instruction skip in order to leak evaluations of the unmasked private polynomial f into the public transcript.

3.3.1 Strategy

Given the private transcript polynomial f and a low-degree random polynomial r, the masked transcript polynomial f' is constructed by:

$$f'(X) = f(X) + (X^n - 1)r(X).$$

In the survey conducted in Table 1 the most common approach to generating evaluations of f' involves generating evaluations of $f(X)$ and $(X^n - 1)r(X)$ separately and adding the results point-wise. By inducing an instruction skip over the function call performing the point-wise addition, the prover will include unmasked evaluations of f in the public protocol transcript. Listing 1 details the assembly-level location of the attack.

Table 3: Polynomial-masking instruction-skip extraction successes out of 30 attempts.

Input-size sweep with $o = 64$											
$n \setminus p_f$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
128	0	2	9	11	14	15	22	18	28	26	30
256	0	5	5	5	11	15	21	19	28	27	30
512	0	2	5	11	12	14	18	24	27	26	30
1024	0	4	8	7	11	17	20	24	22	27	30
Public-opening sweep with $n = 512$											
$o \setminus p_f$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
32	0	2	6	9	7	17	16	21	21	24	30
64	0	5	9	5	10	11	17	19	23	27	30
128	0	2	7	16	13	14	20	21	23	28	30
256	0	5	9	14	12	18	23	25	26	25	30

Listing 1: Instruction skip bypassing polynomial masking.

```
10368: jal   memcpy    # copy unmasked evaluations
       ...             # marker and modulus setup omitted
10374: mv    a0, s8    # output buffer
1037a: mv    a2, s1    # evaluation count
1037c: mv    a1, s2    # masking evaluations
1037e: jal   add_mask  # SKIPPED BY FAULT
       ...             # transcript construction continues
```

3.3.2 Probability of Success

Given the relative simplicity of this attack, in that it requires a single successful instruction skip, the probability of success is the probability of successfully inducing the required fault, p_f:

$$P(\text{success}) = p_f.$$

3.3.3 Experimental Results

The experimental results reflect the relative simplicity of the attack. The results observed in Table 3 are consistent with the predicted probability of inducing the single instruction skip.

3.4 Polynomial Masking Countermeasures

A simple non-equivalence check between the generated evaluations of f and f' would be cheap to introduce, as f and f' should diverge early in the comparison. Furthermore, this attack relies on the point-wise addition modifying f in place. If the masking evaluations were calculated first and then evaluations of f were added to these elements in place, the attack as presented in this section would not work.

3.5 Attacking Interleaved Transcript Padding

The surveyed interleaved padding scheme alternates n private transcript values with n independently sampled cryptographically random values, producing a padded vector of length $2n$. Our attacks against interleaved transcript padding aim to isolate transcript data from random paddings by faulting polynomial interpolation such that the output contains

an unmixed interpolation of the original unpadded private transcript, f. This is achieved by skipping the mixing steps between original and padding elements in Cooley–Tukey. We then use a similar attack during batch polynomial evaluations to maintain the separation when generating evaluations. As such, half of the generated evaluations will be derived only from private transcript data, not random paddings. As a result, when evaluations are revealed during FRI each opening has a $\frac{1}{2}$ probability of revealing private information about the underlying private transcript.

Both faults exploit the divide-and-conquer structure of the Cooley–Tukey algorithm to stop regularly structured values from mixing. Guided by the survey of real-world implementations, Table 2, we will detail attacks against both iterative variants of Cooley–Tukey.

3.5.1 Strategy

The prover begins with the private transcript of length n. Random paddings are interleaved into the private transcript, such that it is of length $2n$. Now the attack specifics diverge for the DIT and DIF variants of Cooley–Tukey. The DIT variant begins with a bit-reverse on its input. This transformation moves the interleaved padding values to the last n positions, effectively turning it into a concatenated padding.

To "ignore" the padding values during polynomial interpolation we must skip mixing across the n-sized blocks for DIT Cooley–Tukey (as the padding values are at indexes $[n, 2n)$) and skip mixing across the 1-sized blocks for the DIF case (as the padding values are interleaved with private transcript values). In order to skip the mixing we induce a single-bit upset on the entry conditions to an inner loop driving the mixing (i.e., on line 5 of Algorithm 1 or line 6 of Algorithm 2, when s is the block size we want to skip we set $k =$ `0x80000000`). Listing 2 illustrates the fault location in assembly in the DIT case.

Listing 2: Single-bit upset on the block index at the entry test of the second Cooley–Tukey loop.

```
        ...                 # outer loop over stages
119e8:  slti  zero,zero,0   # stage head
119ec:  sw    zero,-68(s0)  # i = 0, block index
119f0:  lw    a5,-68(s0)    # reload i
119f4:  bgeu  a5,s6,11d04   # leave block loop if i >= n
                            # FAULT: a5 |= 0x80000000
119f8:  srli  s9,s8,0x1     # stage body, not reached
        ...                 # butterflies, not reached
11d04:  slli  s8,s8,0x1     # len *= 2, later stages run
11d06:  bgeu  s6,s8,118a2   # outer loop back edge
```

In both DIT and DIF cases, if the target mixing step is skipped we produce a corrupted output where the first n elements are an interpolation of only the private transcript elements, f, and the second n elements are a polynomial based on only the random padding values. Next, in order to calculate the Reed–Solomon expansion a further $2n$ zero values are appended to the end of the working vector.

The next stage in the attack involves inducing another fault in Cooley–Tukey, this time while it is used to generate batch polynomial evaluations. This fault again aims to stop the private-transcript-derived values and the padding-derived values from mixing. Again, in the DIT case the working vector is first bit-reversed. Bit reversal places the transcript-derived elements in positions $0, 4, 8, \ldots, 4n - 4$ and the padding-derived elements in positions $2, 6, 10, \ldots, 4n - 2$. As such, skipping the 2-sized block in DIT stops the mixing of padding-derived elements and private-transcript-derived elements. In the DIF case, as no bit-reversal reordering has taken place, n-sized block mixing must be skipped to produce the same effect. These faults are engineered exactly as before, targeting the entry

Table 4: Interleaved-padding extraction successes out of 30 attempts against iterative DIT Cooley–Tukey.

Input-size sweep with $o = 64$											
$n \setminus p_\mathrm{f}$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
128	0	0	1	2	4	10	11	15	15	25	30
256	0	0	2	1	8	8	10	16	19	23	30
512	0	0	2	3	3	4	8	21	22	27	30
1024	0	0	1	2	6	9	9	12	20	26	30
Public-opening sweep with $n = 512$											
$o \setminus p_\mathrm{f}$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
32	0	1	2	3	7	9	13	14	17	27	30
64	0	2	1	1	6	5	7	18	17	26	30
128	0	0	2	3	4	13	6	15	20	24	30
256	0	0	0	4	5	7	11	16	17	28	30

condition of an inner loop when the target block size mixings are to occur.

If both the interpolation and evaluation generation fault are induced successfully, half of the candidate openings for inclusion in the public protocol transcript are unmasked private-transcript-derived values.

3.5.2 Probability of Success

In order for the attack to succeed we must successfully induce a fault at two locations in the STARK protocol, once during polynomial interpolation and once during batch evaluation generation. We must then see a transcript-derived element during the FRI step of the protocol. Given a probability of successfully inducing a single fault p_f and o FRI openings, and approximating openings for FRI as sampled without replacement, we obtain the following probability of success:

$$P(\text{success}) = p_\mathrm{f}^2 \left(1 - \frac{\binom{2n}{o}}{\binom{4n}{o}}\right)$$

3.5.3 Experimental Results

The experimental results in Table 4 and Table 5 support the probability of success presented above. There is a sharp increase in the attack success rate as the probability of successfully inducing a fault moves past the threshold of 0.6 to 0.7. The influence of the number of openings is insignificant, as each opening has a 0.5 probability of leaking private information, so for any reasonable number of openings this component of the probability has little effect.

3.6 Attacking Concatenated Transcript Padding via DIF Recursive Cooley–Tukey

Much like interleaved transcript padding, concatenated transcript padding involves expanding the private transcript to twice its size by adding cryptographically random values. The difference in this case is that the random values are appended to the end of the original transcript. We separate the DIF and DIT variants of this attack due to their difference in methodology.

The process of attacking DIF recursive Cooley–Tukey in conjunction with concatenated transcript padding is conceptually similar to our earlier attacks against interleaved

Table 5: Interleaved-padding extraction successes out of 30 attempts against iterative DIF Cooley–Tukey.

Input-size sweep with $o = 64$											
$n \setminus p_f$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
128	0	0	1	2	5	8	12	17	16	26	30
256	0	0	1	3	4	8	9	12	20	23	30
512	0	1	1	1	5	8	13	15	22	20	30
1024	0	0	1	2	7	7	8	8	20	22	30
Public-opening sweep with $n = 512$											
$o \setminus p_f$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
32	0	1	1	3	5	8	14	21	15	21	30
64	0	1	2	4	3	8	12	10	15	21	30
128	0	0	2	0	6	6	13	18	22	22	30
256	0	1	1	1	4	4	11	13	22	23	30

transcript padding. Much like before, we aim to exploit the structure of Cooley–Tukey to ignore mixing elements in a regular pattern. However, in this case, we attempt three faults, one to stop mixing of blocks sized n during polynomial interpolation, and two more to stop recursive calls in the batch evaluation generation step. Strictly, only one of the two latter faults needs to fire to successfully mount the attack. Each contributes half of the unmasked values in the final output. This results in half of the output elements being directly derived from private transcript data. Unlike previous attacks which leak evaluations of the private polynomial f, this attack directly leaks coefficients of the private polynomial f.

3.6.1 Strategy

The private transcript, A, is padded by appending cryptographically random values increasing its length from n to $2n$. When Cooley–Tukey is invoked for polynomial interpolation, we induce a single-bit upset in the loop counter of the mixing steps before the algorithm recurses (setting $j =$ `0x80000000`, on line 5 of Algorithm 3). This initial mixing would have mixed across n-sized blocks, and thereby combined private transcript values with the padding values. By skipping this mixing step we are able to keep the two separate. After this faulted interpolation the working vector of values is the coefficients of two interleaved polynomials, one generated entirely from private transcript data, the other from only random padding values. At this point the working vector is doubled in size as part of the Reed–Solomon expansion. Listing 3 illustrates the assembly-level location of this phase of the attack.

Listing 3: Single-bit upset on the loop counter at the entry test of the mixing loop, at the first step of the recursion.

```
        ...                 # root call, half = n
11874:  sw   zero,-68(s0)   # j = 0, loop counter
11878:  lw   a5,-68(s0)     # reload j
1187c:  bgeu a5,s5,11d06    # leave loop if j >= half
                            # FAULT: a5 |= 0x80000000
11880:  or   a5,s4,s2       # butterfly body, not reached
        ...                 # a[j], a[j+half] never combined
11d06:  or   a5,s4,s2       # loop exit, on to w2 = root*root
11d0e:  j    11b38          # ... then the two child calls
11cac:  jal  1183c          # each half transformed on its own
```

The next faults target the Cooley–Tukey invocation used for batch polynomial evaluation. We allow root-level mixing to occur. This mixing spreads the private-transcript-derived values and padding-derived values into the zero-padded part of the working vector without mixing the two groups. We then perform an instruction skip over both recursive calls (on lines 11 and 12 of Algorithm 3). This stops any further mixing. Listing 4 illustrates the assembly-level location of this second phase of the attack.

Listing 4: Instruction skips removing both child calls of the first recursion step of the batch evaluation.

```
        ...                 # root butterfly already done
121f8:  mv    a1,s5         # half
121fa:  mv    a0,s3         # a, lower half
121fc:  jal   11d8c         # child 1   SKIPPED BY FAULT 2
        ...
12202:  slli  a5,s5,0x3     # a += half   (child 2 arguments)
12206:  add   s3,s3,a5
12208:  li    a5,1
1220a:  bne   s5,a5,11dbc   # child 2, tail call compiled into a
                            #   loop back edge   SKIPPED BY FAULT 3
1220e:  lw    s2,80(sp)     # call returns
```

Half of the values in the faulted output leak private transcript data. If any of these values are included in FRI openings, private data is leaked.

3.6.2 Probability of Success

It is not completely necessary for the second and third faults to both fire. They each independently preserve private values in a quarter of the output positions. As such, only one is required for a successful attack, and inducing two improves the (already high) probability of witnessing a private value on the public transcript from 0.25 per FRI opening (when only two faults are successfully induced) to 0.5 per FRI opening (when all three faults are successfully induced).

Given the probability of successfully inducing faults 1, 2, and 3 respectively denoted by $p_{\mathrm{f1}}, p_{\mathrm{f2}}$, and p_{f3}, o openings, and n private transcript values, the probability of successfully mounting the attack is given by:

$$P(\text{success}) = p_{\mathrm{f1}}\left(p_{\mathrm{f2}}p_{\mathrm{f3}}\left(1 - \frac{\binom{2n}{o}}{\binom{4n}{o}}\right) + (p_{\mathrm{f2}} + p_{\mathrm{f3}} - 2p_{\mathrm{f2}}p_{\mathrm{f3}})\left(1 - \frac{\binom{3n}{o}}{\binom{4n}{o}}\right)\right)$$

3.6.3 Experimental Results

The experimental results detailed in Table 6. The measured success rates follow the expression above; with a common per-fault probability it reduces to $2p_f^2 - p_f^3$. There is no clear effect of n or o within the tested ranges and sample sizes.

3.7 Attacking Concatenated Transcript Padding via DIT Recursive Cooley–Tukey

The attack targeting DIT recursive Cooley–Tukey diverges significantly from the previous Cooley–Tukey attacks. We are unable to feasibly attack block mixing in this configuration, as such an attack would require n faults, each at the deepest level of recursion. If, for whatever reason, we were unable to induce any one of these faults, the consequently mixed values would contaminate all other output as it propagated through the recursive steps.

As such, attacking concatenated padding via DIT recursive Cooley–Tukey requires a different approach; namely, we attempt to nullify the effects of the padding values by

Table 6: Concatenated-padding extraction successes out of 30 attempts against recursive DIF Cooley–Tukey.

Input-size sweep with $o = 64$											
$n \setminus p_f$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
128	0	1	2	3	8	6	14	18	27	26	30
256	0	1	2	1	4	16	13	18	21	28	30
512	0	1	1	4	6	13	14	18	24	27	30
1024	0	0	2	6	12	11	15	21	22	28	30
Public-opening sweep with $n = 512$											
$o \setminus p_f$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
32	0	0	4	5	6	14	18	14	23	27	30
64	0	1	7	4	8	8	19	20	23	29	30
128	0	0	1	8	6	9	18	24	24	27	30
256	0	0	0	4	7	10	20	20	27	29	30

performing an instruction skip on the return edge of the loop constructing the buffers for the recursive calls, and thereby omitting the cryptographically random padding values from being passed into the recursive calls. This attack requires only a single fault, but assumes that unwritten buffer entries read as zero, e.g., because the buffer is allocated from never-written memory that reads as zero, or because this security-critical code properly zeroises private data before it is freed.

3.7.1 Strategy

To mount this attack we induce an instruction skip on the root invocation of Cooley–Tukey, on the return edge of the loop constructing the buffers, before the recursive calls (the loop on line 4 of Algorithm 4 after $m/4$ iterations where m is the padded transcript size). This jumps out of the loop when the loop reaches the midpoint of the padded transcript. The function then continues as normal, with the padding elements omitted and no longer contributing to the output. Listing 5 illustrates the positioning of the attack at the assembly level.

Listing 5: Instruction skip removing the back edge of the gather that builds the buffers for the recursive calls, at the iteration that has just copied the last private element.

```
        ...                 # root call, half = n
1188c: add  s7,s3,s11       # s7 = a + 16*half, end of input
11890: mv   s8,s3           # s8 = a, walking pointer
11896: lw   a0,0(s8)        # loop top: a[2i]   -> even[i]
118a6: sw   a0,0(s9)
118ae: lw   a0,8(s8)        #           a[2i+1] -> odd[i]
118be: sw   a0,0(s11)
118c6: addi s8,s8,16        # advance one pair
118cc: bne  s8,s7,11896     # loop while s8 != s7
                            # SKIPPED BY FAULT on iteration n/2,
                            #   where s8 = a + 8n, the midpoint
        ...                 # falls through: padding never gathered
```

After the Reed–Solomon expansion and batch evaluation generation $\frac{3}{4}$ of the resulting evaluation positions contain unmasked private transcript-derived values.

Table 7: Concatenated-padding extraction successes out of 30 attempts against recursive DIT Cooley–Tukey.

Input-size sweep with $o = 64$											
$n \setminus p_f$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
128	0	2	4	10	14	14	21	26	25	23	30
256	0	1	6	8	10	13	18	21	26	28	30
512	0	4	6	10	8	13	18	18	25	28	30
1024	0	5	5	10	15	20	19	18	22	27	30
Public-opening sweep with $n = 512$											
$o \setminus p_f$	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
32	0	3	6	7	6	15	18	21	28	24	30
64	0	2	5	7	8	13	21	20	20	29	30
128	0	3	7	5	10	11	14	20	24	29	30
256	0	4	9	6	13	14	14	17	25	29	30

3.7.2 Probability of Success

The probability of success is simply the probability of successfully inducing the single instruction-skip fault. For a probability p_f of successfully inducing the fault:

$$P(\text{success}) = p_f.$$

3.7.3 Experimental Results

Table 7 experimentally validates the direct relationship between the probability of injecting the target fault and the probability of leaking private data into the public protocol transcript.

3.8 Cooley–Tukey Attack Countermeasures

The Cooley–Tukey attacks depend on omitting the butterfly mixing or input-gathering operations while leaving the surrounding control flow intact. Three complementary low-overhead mitigations are available.

The first is a loop (or recursion) counter assertion. At the end of each outer stage, assert that the inner loop counter reached its expected target ($j = m/2$, and ultimately $s = \log_2(n)$), aborting otherwise. The check sits outside the faulted loop and adds no asymptotic overhead to the $\Theta(n \log n)$ butterfly work. A fault model capable of simultaneously suppressing the loop body and advancing the counter to its terminal value is strictly stronger than the single-instruction skip assumed by our attack, and is substantially harder to realise in practice. A similar effect for recursion can be achieved with an accumulator across the recursive calls. This mitigation will not detect input-corruption attacks, such as the attack against concatenated transcript paddings via DIT recursive Cooley–Tukey.

The second is an arithmetic checksum that exploits an algebraic identity of the transform itself. Evaluated at $\omega^0 = 1$, the transform's zeroth output is the sum of its inputs, $y_0 = \sum_j x_j$, or equivalently $m \cdot c_0 = \sum_j x_j$ for an inverse transform of length m carrying the $1/m$ scaling. The prover therefore sums the padded transcript in one independent pass and, after interpolation, compares the result against $2n \cdot c_0$, aborting on mismatch. Modifying a set S of inputs changes the left-hand side of the identity by $\sum_{j \in S} x_j$, a random-looking field element given cryptographically random padding, and each such fault is therefore detected with probability $1 - 1/|\mathbb{F}|$. The cost is $2n - 1$ additions and a single multiplication, negligible against the $\Theta(n \log n)$ cost of the transform itself.

Additionally, the majority of the attacks presented in this work exploit the structural similarity between the padding patterns and the divide-and-conquer designs of Cooley–Tukey. Syncopating padding to some irregular pattern would render this class of attack much more difficult to mount.

3.9 General Countermeasure Against Proof Tampering

In the non-interactive setting, the prover holds the entire proof locally before transmission and can therefore run the STARK verifier on its own output as a final sanity check. Indeed, faults that corrupt the committed witness or transcript are caught with overwhelming probability by the checks involved in the FRI protocol. Since verification is typically considerably cheaper than proving, this mechanism adds only a modest overhead. However, faults injected into the query answers or FRI layer leaves after the commitment can still go undetected.

We emphasise that this countermeasure does not apply to interactive provers, where openings to challenges have already been sent before finalising the entire proof. Per-round consistency checks (e.g., recomputing a Merkle root before releasing it) remain possible, but cannot cover FRI queries that depend on later verifier challenges. Also, this countermeasure checks for proof corruption, and does not detect if the prover has inadvertently created an unmasked-but-valid proof.

4 Conclusion

In this work, we have detailed a set of fault injection attacks against zkSTARK provers capable of violating the ZK property, leaking private information to the verifier. These attacks make use of well-understood fault models, and attack common algorithmic primitives. This work constitutes the first exploration of fault injection attacks specifically tailored to the zkSTARK protocol. To the best of our knowledge, this is the first fault injection work to target general-purpose ZK proving systems.

Acknowledgements

We thank the Cryptography team at Shielded Technologies for their early feedback on this work. Daniel Page was supported in part by Innovate UK via project 10065634 (SCHEME: Safety Critical Harsh Environment Micro-processing Evolution).

References

[0xP] 0xProject. OpenZKP. https://github.com/0xProject/OpenZKP. Git ref inspected via remote HEAD: `c28a5c66b6ee`.

[Azt21] Aztec Protocol. Aztec: Fast Privacy with ZK^2 Rollup, 2021. Technical whitepaper.

[BBHR18a] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *ICALP*, LIPIcs, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[BBHR18b] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *TCC (B2)*, Lecture Notes in Computer Science, pages 31–60, 2016.

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[Eli] Eli Ben-Sasson et al. libSTARK. https://github.com/elibensasson/libSTARK. Git ref inspected: `11c72adf909e`.

[Fac] Facebook / Winterfell authors. Winterfell. https://github.com/facebook/winterfell. Git ref inspected: `2f78ee9bf667`.

[FS86] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO*, Lecture Notes in Computer Science, pages 186–194. Springer, 1986.

[Guia] Guild of Weavers. Distaff. https://github.com/GuildOfWeavers/distaff. Git ref inspected: `fad92ce59292`.

[Guib] Guild of Weavers. genSTARK. https://github.com/GuildOfWeavers/genSTARK. Git ref inspected via remote HEAD: `6172f2bfb6a8`.

[HK24] Ulrich Haböck and Al Kindi. A note on adding zero-knowledge to STARKs. *IACR Cryptol. ePrint Arch.*, 2024:1037, 2024.

[KSV13] Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Trans. Very Large Scale Integr. Syst.*, 21(12):2295–2306, 2013.

[Lam] LambdaClass. Lambdaworks. https://github.com/lambdaclass/lambdaworks. Git ref inspected: `3c8d8f65546c`.

[MAKK24] Puja Mondal, Supriya Adhikary, Suparna Kundu, and Angshuman Karmakar. ZKFault: Fault Attack Analysis on Zero-Knowledge Based Post-quantum Digital Signature Schemes. In *ASIACRYPT (8)*, Lecture Notes in Computer Science, pages 132–167. Springer, 2024.

[Mata] Matter Labs. era-boojum. https://github.com/matter-labs/era-boojum. Git ref inspected: `5a8c4035a9e1`.

[Matb] Matter Labs. Hodor. https://github.com/matter-labs/hodor. Git ref inspected via remote HEAD: `76fc894f199c`.

[Mid26] Midnight Network. Proof Server, Feb 2026.

[MRS25] Shibam Mukherjee, Christian Rechberger, and Markus Schofnegger. Cache Timing Leakages in Zero-Knowledge Protocols. In *AFT*, LIPIcs, pages 1:1–1:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[Plo] Plonky3 authors. Plonky3. https://github.com/Plonky3/Plonky3. Git ref inspected via remote HEAD: `4aed8fe4195d`.

[Pola] Polygon. zkevm-prover. https://github.com/0xPolygon/zkevm-prover. Git ref inspected: 96e02add28c0.

[Polb] Polygon Zero. Plonky2. https://github.com/0xPolygonZero/plonky2. Git ref inspected via remote HEAD: 5d9da5a65bbc.

[RISa] RISC Zero. risc0. https://github.com/risc0/risc0. Git ref inspected via remote HEAD: e459039cc402.

[risb] riscv-software-src contributors. Spike: RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim. GitHub repository, accessed 15 July 2026.

[SCA18] Laurent Simon, David Chisnall, and Ross J. Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *EuroS&P*, pages 1–15. IEEE, 2018.

[Sch80] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980.

[SLP+25] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. Breaking Bad: How Compilers Break Constant-Time Implementations. In *AsiaCCS*, pages 1690–1706. ACM, 2025.

[Staa] StarkWare. ethSTARK. https://github.com/starkware-libs/ethSTARK. Git ref inspected via remote HEAD: 0eb683bfdc9b.

[Stab] StarkWare. Stone Prover. https://github.com/starkware-libs/stone-prover. Git ref inspected: 1414a545e4fb.

[Stac] StarkWare. Stwo. https://github.com/starkware-libs/stwo. Git ref inspected: 5ea059737572.

[Suc] Succinct Labs. SP1. https://github.com/succinctlabs/sp1. Git ref inspected: 3a98ea24252f.

[TBP20] Florian Tramèr, Dan Boneh, and Kenny Paterson. Remote Side-Channel Attacks on Anonymous Transactions. In *USENIX Security Symposium*, pages 2739–2756. USENIX Association, 2020.

[Tri] Triton VM authors. triton-vm. https://github.com/TritonVM/triton-vm. Git ref inspected: 8cd9a0ebe5f6.

[Val] Valida. valida. https://github.com/valida-xyz/valida. Git ref inspected via remote HEAD: 5058de8573e2.