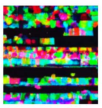

ZKSECURITY

Audit of Zorp - Schnorr signatures over the Cheetah curve and Tip5 hash function

July 18, 2024

Introduction

On July 8th, 2024, zkSecurity was commissioned by Zorp for an audit of their hash function and elliptic curve signature implementation. The audit was conducted by two consultants over a period of two weeks. The audit focused on the correctness and security of the hash function and signature scheme.

The codebase implements the **Tip5 hash function**, which is explicitly designed for use in STARKs. As an algebraic hash function, it operates over elements of a finite field. The field the authors chose, known as the *Goldilocks field*, is $\mathbb{F}_{\mathrm{p}} = \mathbb{Z}/\mathrm{p}\mathbb{Z}$ where $\mathrm{p} = 2^{64} - 2^{32} + 1$.

The codebase also implements the **Cheetah elliptic curve**, which is also designed for use in STARKs. It operates over the same base field, and builds an elliptic curve over the extension field $\mathbb{F}_{\mathrm{p}^6}$.

Finally, the codebase implements a Schnorr signature scheme that uses these two primitives.

Scope

The scope of the audit consisted of a single Hoon files provided by the client:

- `audit.hoon` — This contains the implementations of extension field arithmetic, the Tip5 hash function, Cheetah point arithmetic (in affine, projective, and Jacobian coordinate systems), and a Schnorr signature scheme.

Not in scope but provided as part of the audited code base was a second file:

- `zeke.hoon` - This contains base field arithmetic and other helper functions used in `audit.hoon`, as well as a large amount of functionality unrelated to the audit.

Summary and general comments

Apart from performing a line-by-line audit of the code base, we produced a comprehensive test vector suite for Tip5 and Cheetah, including positive and negative test vectors for Cheetah.

We found, for the values we were able to compare, that the Tip5 hash implementation matched expected outputs. Two small correctness issues were found, one related to edge cases in curve addition, and the other to truncation of hash outputs.

In addition, since a similar but out-of-scope implementation is intended for deployment inside a zkVM, we include informational findings with important security considerations for deployment, including details of constant-timeness, curve arithmetic, choice of curve, and Schnorr challenge hash length.

We believe the Schnorr signature implementation to be secure against forgery, and signatures are *non-malleable*, i.e., modifying a signature makes it invalid. These properties are thanks to the checks

performed on the signature values, namely that the signature scalar is reduced modulo the order ℓ of the Cheetah basepoint.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
00	audit.hoon	Non-unified Curve Addition May Leak Information about Addends	Medium
01	audit.hoon	Some Gates Branch on Potentially Secret Data	Medium
02	audit.hoon	Incorrect Handling of Edge Cases in Curve Addition	Medium
03	audit.hoon	Incorrect Bit Truncation of Hash Outputs	Low
05	audit.hoon	zkVM Code Uses an RNG	Informational
06	audit.hoon	Schnorr Public Keys should be Validated	Informational
07	audit.hoon	Cryptographic Algorithms do not Have Test Vectors	Informational

00 - Non-unified Curve Addition May Leak Information about Addends

audit.hoon

Medium

Description

The `ch-add` gate defined in the `affine` arm branches on whether the provided points are equal up to negation or one of the points is the identity. Depending on these conditions, the gate will either add the points using the usual addition formula, double a point using a doubling formula, or return the identity point. This logic also exists in `ch-add` in the `projective` and `jacobian` arms, and `ch-double` in the `affine`, `projective`, and `jacobian` arms.

Impact

If addition, doubling, and returning the identity take a different number of cycles in the zkVM, then the length of the trace resulting from a `ch-add` call may leak information about which branch was taken, and thus leak information about the addends. If these addends are secret, then this can lead to vulnerabilities.

We note that this, as well as every other constant-time-related finding in this document, is a non-issue if none of the inputs are secret. Since it appears these functions may only used for Schnorr verification of public values in practice, this may indeed be the case.

Recommendation

If indeed these functions operate over secret values, mitigations must be taken. We provide options for mitigations, with tradeoffs:

- Perform both addition and doubling, and perform a constant-time selection operation to pick the correct result (between the sum, doubling, and identity). As a point of reference, this is performed in the `arkworks` ZK framework using the `CondSelectGadget` trait. The caveat to this solution is that the new `ch-add` cycle count is the sum of the cycle counts of the addition and doubling formulas.
- Pad the cycle count to a fixed number, regardless of the branch taken. So long as all cycles appear the same to a verifier, the prover can simply pad out each branch with the correct number of no-op cycles to make the total cycle count invariant under different inputs. The caveat to this method is that this adds a constant that depends on the specific zkVM implementation, and must be kept up to date with the implementation, else a severe privacy lapse can occur.
- Use a different curve over the same base field. The ecGFp5 curve was designed for the same setting as Cheetah, and comes with complete formulas. The downside to this recommendation is that it requires an entirely new curve implementation.

We note that it is not obvious whether unified (i.e., handling addition $\mathrm{P} + \mathrm{Q}$ when $\mathrm{P} = \pm\mathrm{Q}$) formulas for Cheetah exist, let alone complete (handling $\mathrm{P} = \pm\mathrm{Q}$ and $\mathrm{P} = \mathcal{O}$) ones. We note that the Cheetah

Rust implementation claims to use complete formulas, but in fact does not. The formulas they use, from RCB15, do not produce the correct result when $\mathrm{P} - \mathrm{Q}$ is a point of exact order 2.

01 - Some Gates Branch on Potentially Secret Data

audit.hoon

Medium

Description

Adding to finding #00, we find several uses of the `` `?:` `` (branch), `` `?>` `` (assert true), and `` `?<` `` (assert false) runes in the codebase. We list the locations these occur that may branch or assert on potentially secret inputs:

- `` `ch-scal` `` — Similar to finding #00, the Cheetah scalar multiplication gate (in all coordinate types) branches based on the value of individual bits in the scalar.
- `` `f6-pow` `` — In the same way as above, the `` `f6-pow` `` gate branches based on the value of the current bit of the exponent.
- `` `equal:jacobian` `` — This gate asserts that at least one input is not the point at infinity.

Impact

If the length of an execution trace depends on the value of a secret variable, then some information about that secret is leaked. If these variables are Schnorr private keys, which are Cheetah scalars, then this might lead to forgeries.

Recommendation

Again, we recommend that all gates that may plausibly operate over secret data be constant-time. This usually means replacing branching operations with constant-time selection operations. Some assertions can be easily turned into failure conditions, such as in `` `verify` `` on L776: `` `?< =(inf.p-fuzz %.y)` ``. This check can be saved as a boolean, and ANDed with the output boolean, for a constant-time verification.

Some assertions do not occur in a location that has a logical error value, e.g., on L128:

```
++  sbox-layer
  |=  =state
  ^-  (list melt)
  ?>  =((lent state) state-size)
```

In these cases, the conditions must be lifted to a higher-level statement, e.g., "this is the output of the hash function, and the initial state had the correct length". In other cases, where asserts act more like debug statements, such as in the `` `sign` `` gate, they should be removed entirely.

02 - Incorrect Handling of Edge Cases in Curve Addition

audit.hoon

Medium

Description

The implementation of projective addition $P + Q$ on the Cheetah curve starts by handling several edge cases:

- If $P = 0$, return Q
- If $Q = 0$, return P
- If $Q = -P$, return zero
- If $P = Q$, return the result of doubling P

```
++  ch-add
  |=  [p=p-pt q=p-pt]
  ^-  p-pt
  ?:  =(z.p f6-zero)   q
  ?:  =(z.q f6-zero)   p
  ?:  =(q (ch-neg p))  p-id
  ?:  =(p q)  (ch-double p)
```

The problem with the checks for $Q = -P$ and $P = Q$ is that they don't use the correct notion of equality for projective points: Namely, two points are equal if one can be obtained from the other by scaling all coordinates by $\lambda \neq 0$:

$$(x, y, z) \equiv (\lambda x, \lambda y, \lambda z)$$

However, the code only checks for exact equality of all coordinates using Hoon's built-in equality operator `=`.

Consequently, cases where P and Q are equivalent but not equal receive no special handling by this code.

The subsequent projective addition formulas do not handle the $P \equiv Q$ case correctly; they return $(0, 0, 0)$ as the result.

In contrast, the $P \equiv -Q$ case is already handled correctly, a point at infinity $(0, y, 0)$ is returned but with $y \neq 1$.

The same problem exists for addition in Jacobian coordinates. Again, naive direct equality is used instead of checking for equivalence classes:

```
?:  =(q (ch-neg p))  j-id
```

```
?:  =(p q)  (ch-double p)
```

Impact

It appears that the $\mathrm{P} \equiv \mathrm{Q}$ edge case cannot occur in the double-and-add algorithm used for scalar multiplication. It is theoretically possible to be reached in the verification of Schnorr signatures but seems hard to exploit.

Nevertheless, the incorrect edge case complicates the security analysis of signatures and presents a liability in the code base that could become toxic when point addition is used for other purposes.

Recommendation

One possible fix is to implement a dedicated function that checks for equality of two projective points, and use that in place of the `=` operator. For Jacobian points, such a function already exists.

However, there is a more efficient solution. Projective addition computes as a byproduct values `a`, `b` that make it easy to determine both edge cases:

```
++  ch-add
  |=  [p=p-pt q=p-pt]
  :: [ZKSECURITY] ... elided ...
  =/  a   (f6-sub (f6-mul y.q z.p) (f6-mul y.p z.q))
  =/  b   (f6-sub (f6-mul x.q z.p) (f6-mul x.p z.q))
```

Note that $\mathrm{b} = 0$ iff $\mathrm{x_P}/\mathrm{z_P} = \mathrm{x_Q}/\mathrm{z_Q}$, and $\mathrm{a} = 0$ iff $\mathrm{y_P}/\mathrm{z_P} = \mathrm{y_Q}/\mathrm{z_Q}$. If both $\mathrm{a} = \mathrm{b} = 0$, the points are equal and we should double. If $\mathrm{b} = 0$ but $\mathrm{a} \neq 0$, the points must be negations of each other and we can return the point at infinity.

The following changes correct the original code to work in all cases without any loss of efficiency:

```
- ?:  =(q (ch-neg p))  p-id
- ?:  =(p q)  (ch-double p)
  =/  a   (f6-sub (f6-mul y.q z.p) (f6-mul y.p z.q))
  =/  b   (f6-sub (f6-mul x.q z.p) (f6-mul x.p z.q))
+ ?:  =(b f6-zero)
+     ?:  =(a f6-zero)  (ch-double p)  p-id
```

Likewise, in the Jacobian addition the variables `e` and `f` can be tested for zero to determine edge cases.

03 - Incorrect Bit Truncation of Hash Outputs

audit.hoon

Low

Description

There is a mistake in `trunc-255`, which truncates 4 or more base field elements to 255 bits:

```
++  trunc-255
  |=  a=(list belt)
  ^-  @
  ;:  add
    (snag 0 a)
    (mul p (snag 1 a))
    :(mul p p (snag 2 a))
    :(mul p p p q:(dvr (snag 2 a) (bex 63)))
  ==
```

The last line uses `(snag 2 a)`, the element at index 2, for a second time; instead of `(snag 3 a)` which would be correct.

As a result, instead of using about 255 bits from the input list, the code only uses about 192, i.e. the first 3 elements each of which contributes almost 64 bits, and repeats the third element's bits.

The function is called by `hash-varlen-trunc-255`, where the effect is to use fewer bits of the hash output than intended.

Impact

Shortening the hash output makes it easier to find preimages and collisions. 192 bits still leaves plenty of security for preimage resistance against brute-force attacks, but the security level for collision resistance is reduced to 96 bits.

The specific purpose of `hash-varlen-trunc-255` is to be a component of the Schnorr signature algorithm for computing the challenge. In this context, collision resistance is not needed to protect against forgery of honest signatures. The analysis by **Neven et al.** shows that 128 hash output bits should be sufficient to keep the security level at 128.

Recommendation

Fix the typo by changing `(snag 2 a)` to `(snag 3 a)`.

05 - zkVM Code Uses an RNG

audit.hoon

Informational

There are two instances in `audit.hoon` that an algorithm makes use of a random number in a given range, specifically invoking `(rad og eny)`. These are in signing and key generation. In general, a random number generator is not callable from a circuit, since all inputs to the proof must be known in advance. In other words, for these gates to be translated to a zkVM system, they must be made deterministic, e.g., by taking their randomness as a ZK witness. We also note that poor sources of entropy may lead to concrete attacks, e.g., when sampling a nonce for Schnorr signing.

It is not common for a signature to be computed in zero-knowledge, but if this is indeed needed, and an analogue of the Schnorr `sign` gate appears in the zkVM, then it must take as a witness the value of the random nonce `s-fuzz`. We recommend, following the design of Ed25519, that this step be done deterministically, using a high entropy secret seed k, and computing $\mathsf{sfuzz} = \mathrm{H}(\mathrm{k}||\mathrm{M})$. This computation can be performed outside of the circuit, and so a secure wide-output hash function such as SHA-512 can be used for H. This adheres to the guidelines in 4.1.1 Algorithm 2 of BSI TR-03111 that the digest bitlength be 64 bits larger than the bitlength of the target space.

We recommend a similar practice for the `keygen` gate, though we do not expect this to appear in any zero-knowledge deployment.

06 - Schnorr Public Keys should be Validated

audit.hoon

Informational

As `audit.hoon` does not contain any deserialization logic, we do not know how Schnorr public keys are currently deserialized from their wire representation. We include this informational finding to point out some useful validation steps that should be done on public keys before using them to verify signatures.

Firstly, a verifier should check that the provided public key is on the curve. Without this check, we have no guarantees about the behavior of curve arithmetic, or unforgeability of signatures. This may be fine in some cases—a user is free to choose weak keys for themselves as long as it doesn't hurt anyone else—but it is probably helpful to ensure that it is on the curve. We note that, depending on use case, this check can be done natively by the zero-knowledge prover, and does not have to be encoded in the zkVM.

Some Schnorr implementations do further validation steps, including checking that points are *canonically encoded*, i.e., have coordinates that are represented using the fully reduced form in the field. We recommend this check, but note that it is not clear if there are any attacks if it is omitted. Such non-canonically encoded public keys and signatures (for Schnorr signatures that include a curve point rather than the digest e) are considered in detail by **DeValence**. Again, depending on use case, this check can be done out-of-circuit.

We also take the time to point out another scenario that can be prevented by further validation. We include this for completeness, and note that, for the specified use cases, it does not appear that Zorp requires resiliency to this sort of attack. A *key substitution attack* is an attack wherein a new public key can be created such that it verifies a pre-existing message-signature pair.

It is possible to perform a key substitution on the current Schnorr implementation. Suppose the public key Q is an element of the prime-order subgroup, and (e, s) is a signature on m, i.e., $\mathrm{e} = \mathrm{H}(\mathrm{m}, \mathrm{sG} + \mathrm{eQ})$. If e is a multiple of one of the smaller subgroup orders $\mathrm{c} \in \{2, 5, 29, 181, 155833, 866216795937074724496864723 61\}$, then $\mathrm{e} = \mathrm{H}(\mathrm{m}, \mathrm{sG} + \mathrm{e}(\mathrm{Q} + \mathrm{Q}'))$, where Q' is a non-zero element of order c. In other words, the signature (e, s) on m is validated under a different public key $\mathrm{Q} + \mathrm{Q}'$.

Key substitution attacks can be prevented by either asserting that all public keys are *torsion-free*, i.e.., have no low-order components. Another way to prevent this is to include the public key in the challenge hash computation (see section 5.4 of **BCJZ20**). This has the added benefit of improving *multi-user security*, i.e., forgery security against an attacker whose goal is to forge under *any* of a large set of public keys (section 5.3).

07 - Cryptographic Algorithms do not Have Test Vectors

audit.hoon

Informational

Description

The Tip5 and Cheetah implementations do not have tests which validate them against known test vectors. At Zorp's request, we produced two Sage scripts, `tip5.sage` and `cheetah.sage`, which produce test vectors for these primitives.

`tip5.sage` outputs a Hoon file containing input-output pairs for both the fixed- and variable-length-input Tip5 hash function. The test vectors each have a *truncated output*. This value is obtained by treating the first 4 field elements of the digest as a little-endian encoding of a base-p integer (where p is the prime field modulus), and truncating to 255 bits. This follows the BSI TR-03111 recommendation in section 4.2 to truncate hash functions to the bitlength of the generator's order.

`cheetah.sage` outputs two Hoon files, one containing positive test vectors—those which include outputs that are considered correct—and negative test vectors—those which should cause an error. The positive test vectors contain known-answer tests for Cheetah affine point addition, which includes random point addition, point doubling, and adding two points whose difference is a point of exact order two (this is an edge case in the RCB15 formulas). The other vectors test multiplying a randomly chosen point with a randomly chosen scalar and multiplying the Cheetah basepoint with a randomly chosen scalar.

Negative test vectors include invalid representations of points and scalars. Invalid scalars, represented as integers, are those which are greater than or equal to the order ℓ of the Cheetah basepoint, i.e., not reduced modulo ℓ. Invalid points, which we represent with affine coordinates, may be invalid in two ways. First, the point may simply not lie on the curve. Second, it may lie on the curve but have coordinates which are not properly reduced. An unreduced point may be unreduced in the degree-6 extension field, i.e., all coefficients are reduced modulo p, but the extension field element has degree greater than 6 (in other words, the number of coefficients is greater than 6). An unreduced point may also be unreduced in its coefficients, i.e., all extension field elements have the correct degree but a coefficient is greater than or equal to p. All these cases are included in the negative test vectors.