SKALE

Smart Contracts. Audit

Mikhail Vladimirov and Dmitry Khovratovich

4th November 2020

This document describes the audit process of some SKALE smart contracts performed by ABDK Consulting. We did not review all the contracts in the repo, but only the subset listed below. We also did not review the cryptographic protocols used in SKALE, particularly the threshold-signature based consensus.

1. Introduction

We've been asked to review a subset of SKALE smart contracts given in separate files by tag 1.5.2-develop.21.

2. Decryption.sol

In this section we describe issues found in the Decryption.sol.

2.1 Minor Issues

- This contract contains only utility functions that doesn't modify blockchain state. It should be turned into a library, and all its function should be made internal.
- The contract name is confusing, as the contract contains functions for both, encryption and decryption. Better name would be "`Crypto`".
- Lines 35,42 The name is confusing, as one could think that this parameter contain encryption key (secret number). Better name would be "`plaintext`".
- Lines 35,42: This function doesn't have to be external. as it doesn't access transaction parameters. Consider turning it into public (or internal when used as a library).
- It should be warned that the encryption functionality in this contract is secure for one-time keys only.

3. ECDH.sol

In this section we describe issues found in the ECDH.sol.

The code of this contract is based on experimental (acc. to its author) code https://github.com/jbaylina/ecsol/blob/master/ec.sol . It has many issues (see below) It should not be used as is. We recommend using a different code such as https://github.com/tdrerup/elliptic-curve-solidity.

3.1 Major Flaws

This section lists major flaws, which were found in the smart contract.

- This contract methods do not do any sanity check for their inputs. They can be called with non-points and produce incorrect results.
  OUTCOME: PROBABLY NOT AN ISSUE AS LONG AS THE CALLER DOES NECESSARY CHECKS
- ~~The function does not do any range check, though it probably makes sense only for `a<_N`. For example, for `a=_N` the function returns 1, which is probably wrong. Also, it returns 0 for `a=0` which also doesn't seem right.~~
  OUTCOME: FIXED

3.2 Moderate Flaws

This section lists moderate flaws, which were found in the smart contract.

1. ~~For transparency there should be a formula of how this generator point was selected~~
   OUTCOME: Provided
2. ~~In regular homogenous representation there is no point starting with (0,0). Usually the identity point is represented as (0,1,0).~~
   OUTCOME: Not an issue

3.3 Minor Issues

- Should be “^0.6.0” according to common best practice, unless there is something special about this particular version.
- This contract could be turned into a library, and its functions could be make internal, which will make it cheaper and more convenient to use from other smart contracts.
- It is really necessary to have zero named constant? Just removing it from the formulas would make code simpler and more efficient.
- This function has very much in common with “`deriveKey`” function and should delegate to it or somehow share code with it.
- This function has very much in common with “`jAdd`” function and should delegate to it or somehow share code with it.

- This computes `z1z2` yet another time.
- It is unclear which formulas are used here. There exist complete formulas for addition in homogenous coordinates with fewer operations: https://eprint.iacr.org/2015/1060.pdf page 12.
- Most addition laws have special doubling laws that are faster.

4. FieldOperations.sol

In this section we describe issues found in the FieldOperations.sol.

4.1 Major Flaws

1. ~~This library mixes G1 operations and F^2 operations as extension field over F. This is confusing and error-prone, e.g. zero elements are different in those objects. Consider extracting non-common code into a derived contract.~~
   OUTCOME: FIXED
2. ~~This produces `p` if `s.a+d.a=p`~~
   OUTCOME: FIXED
3. ~~As in many places around this code, this can happen to be `p`. Such value can be used internally but probably should not be returned to the caller~~
   OUTCOME: FIXED
4. ~~This function seems to produce incorrect output when the result is the point at infinity. We could not verify that the code is the same as in (alleged) https://github.com/scipr-lab/libff/blob/master/libff/algebra/curves/alt_bn128/alt_bn128_g2.cpp#L202-L212 Those are two different algorithms. Consider using a closer code or demonstrate correctness of the new one.~~
   OUTCOME: It was demonstrated that the code matches the old one

4.2 Moderate Flaws

1. ~~Name is confusing. Is it the G2 generator? If so, it should be demonstrated or a link should be given in the code.~~
   OUTCOME: FIXED

4.3 Minor Issues

1. Should be "^0.6.0" according to common best practice, unless there is something special about this particular version.
2. This library should be in its own file named "Fp2Operations.sol".
3. It is unclear how `Fp2` is defined exactly. Documentation is needed.
4. Names "`value1`" and "`value2`" are longer than "x" and "y", often used in similar cases, but aren't more descriptive.
5. This function could be rewritten as:

```
return F2Point({
a: addmod (d.a, P - s.a, P),
```

```
b: addmod (d.b, P - s.b, P),
});
```

6. This function could be rewritten as:

```
return F2Point({
a: addmod (
mulmod (v1.a, v2.a, P),
P - mulmod (v1.b, v2.b, P),
P),
b: addmod (
mulmod (v1.a, v2.b, P),
mulmod (v1.b, v2.a, P),
P)
});
```

7. This function can be rewritten as:

```
return F2Point({
a: addmod (
mulmod (v.a, v.a, P),
P - mulmod (v.b, v.b, P),
P),
b: mulmod (v.a << 1, v.b, P)
});
```

8. This could be rewritten as: `uint t2 = addmod (t0, t1, p);`
9. For constant power, constant 256-bit modulo, and 256-bit base, unrolled exponentiation by squaring could be cheaper than using precompiled contract.
10. This library should be in its own file named "G2Operations.sol".
11. Name is confusing. Is it the G1 generator? If it is a generator, it should be demonstrated or a link should be given in the code.
12. Not obvious that it belongs to the group.
13. This function differs from "`isG1Point`" only in how arguments are packed. Probably one of these two functions is redundant. The same for `isG2` and `isG2Zero`.
14. The code after this closing brace looks like it is always executed, while it is executed only if `(x, y)` is not G2 zero point. Consider putting the rest of the function into explicit "else" branch.
15. Just to compare with `TWISTB`, instead of subtracting it and comparing with zero, would make code simpler and more efficient.
16. 3 should be a named constant.

5. FractionUtils.sol

In this section we describe issues found in the FractionUtils.sol.

5.1 Moderate Flaws

1. ~~This function does not do any range check though probably not all inputs are valid, for example zero denominators.~~
   OUTCOME: Issue is false
2. ~~Here an intermediate overflow is possible when multiplication overflows but the reduced fraction does not overflow. Consider reducing fractions `a.n/b.d` and `b.n/a.d` first.~~
   OUTCOME: Added an extra constraint on the inputs

5.2 Minor Issues

1. Should be "^0.6.0" according to common best practice, unless there is something special about this particular version.
2. Making both, numerator and denominator 128 bit would make it possible to fit the whole fraction into a single 256-bit word. Also, operations would be simpler and more efficient.
3. This implementation performs precise reduction, which is quite gas consuming and doesn't make much sense, as it doesn't guarantee that numerator and denominator will become small. Not sure what the fractions are supposed to be used for, but probably approximate reduction, that ensures, that both, numerator and denominator fit into certain number of bits, say 128, whodle be cheaper and much more helpful.
4. Should be:
   ```
   (_a, _b) = (_b, _a);
   ```
5. `_a % _b` would be cheaper as we just checked that `_b` is not zero.

6. KeyStorage.sol

In this section we describe issues found in the KeyStorage.sol.

6.1 Moderate Flaws

1. This function does not do validity check for the current key in progress, whereas it may not be uninitialized.
2. This function can be called multiple times. Overall, the new key initialization-adding-finalize workflow can be started at any step, may skip steps, does not log any event. This is error-prone.

6.2 Minor Issues

1. Should be "^0.6.0" according to common best practice, unless there is something special about this particular version.
2. This contract probably should log some event.

3. It should be warned that this code is not suitable for BLS key aggregation due to rogue key attacks.
4. This function would be unnecessary if "`_schainsPublicKeys`" storage variable were public.
5. This and the following function would be unnecessary if "`_previousSchainsPublicKeys`" storage variable were public.
6. This method's functionality is different from checking that the G2 element is a zero element as in `` `isG2Zero()` ``. Maybe it checks for initialization?
7. These functions are not used.
8. This function does not do any check on the current public key contrary to `` `finalizePublicKey` ``.

7. MathUtils.sol

In this section we describe issues found in the MathUtils.sol.

7.1 Minor Issues

1. Should be "^0.6.0" according to common best practice, unless there is something special about this particular version.
2. Is this event really needed in a deployed contract? How would a user make use of it?
3. What about this? `return a > b && a - b > _EPS;`
4. In case `a == b + _EPS`, `a` is neither "significantly greater" than `b`, nor "approximatelyEqual" to `b`. Is this fine?

8. Precompiled.sol

In this section we describe issues found in the Precompiled.sol.

8.1 Minor Issues

1. Should be "^0.6.0" according to common best practice, unless there is something special about this particular version.
2. Lines 39, 54, 92: "`gas()`" would be cheaper in terms of both bytecode size and execution cost. Also `mul(6,0x20)` should be precomputed.
3. The curve is usually called BN254 to avoid confusion with BN curves with 256 bit primes.
4. It should be called `` `verifyPairing` ``.

9. Schains.sol

In this section we describe issues found in some functions of Schains.sol as we were asked to review only a few of them.

9.1 Critical Flaws

1. ~~For the BLS protocol to be secure, the hash-to-curve function should behave like a random oracle. It is allowed to use a regular hash + counter to get into a valid x coordinate, but the counter must not be arbitrary otherwise any point can be obtained. This function should do a range check on the counter.~~
   OUTCOME: FIXED

9.2 Moderate Flaws

1. ~~These parameters are not range checked to be field elements. Consider adding explicit checks.~~
   OUTCOME: FIXED

9.3 Minor Issues

1. Only the hash of this value is used, so it would be cheaper to pass the hash instead of the value.
2. `abi.encodePacked(schainName)` is equivalent to `bytes(schainName)`.

10. SkaleDKG.sol

In this section we describe issues found in some functions of SkaleDKG.sol as we were asked to review only a few of them.

10.1 Critical Flaws

1. ~~An X-coordinate of a DH-derived shared keypoint is not a uniform 256-bit value. Moreover, it probably has its upper bit zero, and some other higher bits biased. An encryption on such a key value will leave upper bits of a 32-byte plaintext untouched. A proper way is to hash the coordinate with a 256-bit hash function first.~~
   OUTCOME: FIXED

10.2 Moderate Flaws

1. ~~This will be `P` if `b==0`.~~
   OUTCOME: FIXED

10.3 Minor Issues

1. As public keys are group elements, it is more appropriate and clear to store them as integers or Fp2.
2. Intermixing type uint256 with its alias uint in the same contract makes code harder to read. Use consistent type names.
3. This function may return unexpected results on non-curve inputs. Is this OK?
4. There should be a call to some function that outputs a neutral G1 element.

11. SkaleVerifier.sol

In this section we describe issues found in SkaleVerifier.sol.

11.1 Critical Flaws

1. ~~For the BLS protocol to be secure, the hash-to-curve function should behave like a random oracle. It is allowed to use a regular hash + counter to get into a valid x coordinate, but the counter must not be arbitrary otherwise any point can be obtained. This function should do a range check on the counter.~~
   OUTCOME: FIXED

11.2 Moderate Flaws

1. ~~This parameter is not range-checked though not all values are valid. Some values are silently taken mod p. This could probably be used for malleability-kind attacks.~~
   OUTCOME: FIXED
2. ~~For zero `.b` we have `newSignB=P` which is probably wrong. Probably, suspicious condition above tries to address this issue, but this only works in case `b==0` implies `a==0`. It would be better to have a separate function to safely invert field element, something like `(P - x % P) % P`. Or, in fail-fast manner:~~

   ```
   require (x < P);
   return (P - x) % P;
   ```

   OUTCOME: FIXED

11.3 Minor Issues

1. There is no access control checks in this contract, why does it inherit from “Permissions”?
2. This condition is suspicious. Which part of the signature verification does it correspond to?
3. The outcome if this check is predefined in case “`else`” branch of the conditional statement above was executed. Consider moving into the “`then`” branch.

4. This was verified in the `_checkHashToGroupWithHelper` and is probably redundant.
5. This parameter seems to be redundant.
6. The cheapest checks usually go first in conjunction/disjunction, but here they go last.

12. StringUtils.sol

In this section we describe issues found in StringUtils.sol.

12.1 Minor Issues

- This variable is used without being initialized.

13. Summary

We recommend the following:

1. Fix critical and major flaws.
2. Check that all the libraries do the necessary range checks.
3. Outline the cryptographic protocol as a companion to the contracts.
4. Pay attention to moderate flaws.
5. Check minor issues.