Security review of CTHelpers.sol

Federico Giacon Alexander Herrmann

December 2019

Contents

1 Introduction

This document is a formal audit of the functionalities offered by the contract `CTHelpers.sol`, an essential tool in the conditional tokens contract.

The audit does not point out any critical security issue in the Solidity code. However, it puts to question the current implementation of the elliptic curve multiset hash function that is used to create collection IDs. The current implementation differs from the one described in the scientific paper it was derived from for efficiency reason. There is no proof that the new implementation matches the security guarantees provided by the original construction. To address this issue, we propose minor changes to the current construction that make the hash function provably secure yet still efficient to use in a Solidity contract.

Other minor remarks on the code are also raised.

We begin our review with section 2, a high-level overview of the conditional token framework using the CTHelpers library. Next, in section 3, we give a high-level description of the code in `CTHelpers.sol`. A detailed description discussing implementation choices is found in section 4. We present our findings and recommendations in section 5. Finally, we point out a particular behavior of the conditional tokens in section 6 which, while not directly pertinent to the code in `CTHelpers.sol`, might still be of interest to auditors of the conditional tokens contract.

2 High-level description of the contitional tokens parent contract (`ConditionalTokens.sol`)

This contract provides a framework to create and manage *conditional tokens*: Tokens that are tied to the outcome of one or more real-life conditions.

Conditions. A condition is characterized by an *oracle* and the number of possible outcomes. Each condition is identified by a unique condition ID, and its outcomes are intended to be mutually exclusive. The oracle is tasked with settling the outcome of the condition: This can be either selecting a single final outcome or assigning a "weight" to each possible outcome, intuitively representing to which degree each outcome came to pass. When the oracle resolves the condition, this resolution cannot be revoked nor changed.

Conditional tokens. Each user can tie an amount of *collateral tokens* (which can be any ERC20 token) to some condition. To this end, the user selects a partition of all possible outcomes of the condition and *splits* the collateral token into new conditional tokens, one for each set of outcomes in the partition. When taken together, all these tokens cover all possible outcomes of the original condition. In similar fashion to ERC20 tokens, each of these tokens can then be transferred to other users and other users can be authorized to transact on behalf of the owner. When a condition is settled, the user can convert conditional tokens back into collateral tokens, retrieving an amount proportional to the weights of the outcomes associated to each token.

A token associated to multiple outcomes of the same condition can be split into multiple tokens based on any arbitrary partition of these outcomes. Multiple tokens associated to disjoint outcomes of the same condition can be merged together into a single token associated to the union of all outcomes; if this set contains all outcomes of a condition, a corresponding amount of collateral token are recovered.

Multiple conditions. Each conditional token can be further tied to other conditions, resulting in the creation of a corresponding amount of different conditional tokens. A token associated to multiple conditions is functionally equivalent to a token associated to a single one. If any of the conditions comprising a token is settled, its owner can recover an amount of conditional tokens proportional to the weight of the associated outcomes of this condition. The resulting token includes all conditions in the original token except the one which was settled.

Collections and positions A *collection* is a list of conditions and, for each condition, corresponding outcomes. A collection is uniquely identified by its collection ID. A *position* is a collection associated to a (single) collateral token. Again, a position is uniquely identified by its position ID. The position ID serves as the token identifier for each conditional token generated by the contract. This implies that each conditional

token is associated to a unique collateral token. The collateral token cannot be changed without withdrawing the collateral.

3 High-level description of the library (`CTHelpers.sol`)

This contract is a support library that manages creation and updating of the various identifiers used in the conditional tokens contract. The main purpose is to isolate these more technical functions into a separate library, where they can be implemented separately from the logic of conditional tokens. Two cryptographic routines are used by this library, namely `keccak256` and the elliptic curve `alt_bn128`, which are used to implement the multiset hash function described by Maitin-Shepard et al. [5].

- `getConditionId` takes as input the address of an oracle, an arbitrary 32-byte integer value, and the number of possible outcomes. The output is a unique identifier for the condition described by the input values.
- `sqrt` takes as input an element X in the finite field $\mathbb{F}$ over which the elliptic curve `alt_bn128` is defined. If X is a square in $\mathbb{F}$, this function outputs one of the square roots of X in $\mathbb{F}$. Otherwise, it returns an element of $\mathbb{F}$. (In this latter case, the properties of the returned element are not relevant, except that it is not a square root of X.)
- `getCollectionId` takes as input a collection ID (`parentCollectionId`), a condition ID, and a set of outcomes. The output is a collection ID characterized by all pair condition/outcomes associated to `parentCollectionId` and additionally the pair condition/outcomes given as input. Intuitively, this function outputs a new collection ID by restricting `parentCollectionId` to the outcomes of a further condition.
- `getPositionId` takes as input the address of a collateral token contract and a collection ID. It outputs a position ID.

4 Detailed description of the contract (`CTHelpers.sol`)

The contract defines two constants, P and B. These numbers are part of the description of the elliptic curve `alt_bn128`, which was introduced with the Ethereum Improvement Proposal (EIP) 196 [6]. P is the prime number corresponding to the size of the underlying finite field $\mathbb{F}$, while B is the constant term in the equation defining the elliptic curve: $Y^2 = X^3 + B$.

- `getConditionId` hashes together all its input values with `keccak256` to create unique identifiers for each condition ID.
- `sqrt` uses field multiplications modulo P, stated in assembly code, to compute square roots in the finite field $\mathbb{F}$. Since $P = 3 \mod 4$, a square root of an element $X \in \mathbb{F}$ can be computed as $Y \coloneqq X^{\frac{P+1}{4}}$ (see appendix A). The assembly code of this function uses modular multiplications to execute this one exponentiation operation.

  The exponent is the same for every input to the function, and the sequence of multiplications is therefore optimized for this particular exponent. This sequence is obtained starting from an *addition chain* summing to $\frac{P+1}{4}$.[1] The addition chain in the exponent is translated into a multiplication chain in the finite field.

  Using multiplication chains is more efficient than the more naïve technique of squaring and multiplying: The current implementation requires 323 finite-field multiplication, while squaring and multiplying

[1] An addition chain is a sequence of elements $x_0, x_1, \ldots, x_n$ such that $x_0 = 1$, $x_n = \frac{P+1}{4}$, and every element x_i can be written as the sum of two elements of the sequence with index smaller than i.

requires 363 multiplication. A short simulation confirms that square and multiply costs more gas than the current implementation, even despite the fact that square and multiply requires less memory. The gas cost of the current implementation is also lower than that of executing the precompiled contract `0x05`, which implements modular exponentiation.

The low gas cost makes it worthwhile to check if an element x is a square in the finite field by running this function on x, squaring the result and comparing it to x, as done in `getCollectionId`. This is in fact more gas efficient than the theoretically more efficient technique of computing the quadratic residue using the properties of the Jacobi symbol.

- `getCollectionId` generates unique identifier for collection IDs. The identifier is computed using an elliptic curve multiset hash function [5]. This is a hash function H with the property that, given any two multisets A, B, $H(A \cup B) = H(A) + H(B)$. `getCollectionId` takes three input values: `parentCollectionId` is a hash value for a collection ID, while `conditionId` and `indexSet` form, together, a pair representing a commitment to a condition. The output of the function `getCollectionId` is the hash of the (multi)set represented by `parentCollectionId` with the added pair for the condition.

  The function works as follows: First the hash of the pair (`conditionId`, `indexSet`) is computed, obtaining an elliptic curve point. Then the hash is added to the elliptic curve point corresponding to `parentCollectionId` and the result is returned.

  The hash of an element *pair* = (`conditionId`, `indexSet`) is computed as `mapToGroup`(`keccak256`(*pair*)), where the function `mapToGroup` converts the output bitstring of `keccak256` into an elliptic curve point. This function is implemented by setting the x coordinate to be the input bitstring and incrementing it by one until a valid y is found such that (x, y) is a valid curve point. Heuristically, it is expected that this sampling method takes in a single run at most about 254 attempts to find a valid y in the worst case (see appendix B), on average much less often (a single time). One bit of the hash is also used to decide which of the two possible y values to choose.

  The addition in the elliptic curve is executed by running the precompiled contract `0x06`. The contract `CTHelpers.sol` takes care of converting the elliptic curve points from long notation (x and y) to short notation (x and sign bit) and vice-versa.

- `getPositionId` hashes together all its input values with `keccak256` to create a unique identifier for position IDs.

5 Comments on (`CTHelpers.sol`)

- `sqrt`

  1. [*category: readability*] The documentation of this function lacks a key detail to understand what is happening: Computing a square root of a square X in a finite field of order P is equivalent to computing the field exponentiation $X^{\frac{P+1}{4}}$. The addition chain is then just a tool to efficiently compute this exponentiation. We recommend to update the comment text.
  2. [*category: readability*] The assembly code in this function could be replaced by a singe call to the precompiled contract `0x05` implementing modular exponentiation. This change would require about 7.000 extra gas for each call to the function. It might be worth to accept the additional cost in exchange for easier maintenance and readability.

- `getCollectionId`

  1. [*category: security, major*] The code implements the elliptic curve multiset hash function described by Maitin-Shepard et al. [5], but uses a different map from bitstrings to elliptic curve points from the one specified in the paper. The change is motivated by the fact that the original map is not efficiently implementable in the Ethereum Virtual Machine (EVM). The replacement map

of choice is the "try-and-increment", described by Icart [4]: While no concrete attacks on this implementation were found during the auditing process, it is also not clear why this map would be a secure choice in this context. The security proof by Maitin-Shepard et al. [5] does not apply to this modified construction, making the decision to use this specific construction moot. In fact, the major contribution of their paper is the very specific choice for the map from bitstrings to elliptic curve points.

We recommend to use instead the similar map "increment-then-hash", which at a high level of abstraction works as follows:

```
counter = 0
repeat:
  x = keccak256(bitstring, counter)
  counter = counter + 1
until there exists y such that (x, y) belongs to the curve
```

This construction modifies the multiset hash function in a way that keeps it efficient to implement on the EVM and at the same time gives a formal proof of its collision resistance (see appendix C).

2. [*category: security, minor*] The last bit of the input `parentCollectionId` is ignored.[2] This is because of line 408. It follows that there are distinct values of `parentCollectionId` that yield the same output each time the remaining input values are the same. We recommend to sanitize the function's input so that there is no more than one one valid parent collection ID representing the same concept, for example by requiring `parentCollectionId` to be an integer smaller than P plus the sign bit (short form of an elliptic curve point).[3]
3. [*category: security, minor*][*line: 393, 394, 398, 402*] The hash output is used to generate a random value `x1`, and its last bit `odd` is also used to determine the parity of its y coordinate. The same randomness (the bit `odd`) should not be used for two different purposes to avoid spurious correlations between unrelated values. As a solution, we recommend to truncate `x1` to 255 digits.
4. [*category: efficiency*][*line: 402*] The code uses the bit `odd` to determine the parity of the point. Since this bit is uniform, the output distribution does not change by replacing the conditions in the `if` by the simpler test `if(odd)`. We recommend to simplify the `if` condition and consequently to rename the variable `odd` to highlight it has lost its semantic value. However, this would require to change some hardcoded values in the tests.
5. [*category: efficiency*][*line: 411*] The condition can be rewritten as `if(odd != (y2 % 2 == 1))`. Similar change can be made in line 402, assuming the change recommended in comment 4 is not carried out.
6. [*category: readability*] The code is not clear for a reader who is not familiar with the inner working of elliptic curves. We recommend to create intermediate functions to improve readability such as:

```
function ecGetCandidateY(uint hash) private pure returns (uint)
function ecFromShortToLong(uint short) private pure returns (uint, uint)
function ecFromLongToShort(uint x, uint y) private pure returns (uint)
```

6 Negative and multiple tying to condition outcomes

The auditing process drew attention to some unexpected behaviors of the conditional tokens parent contract (`ConditionalTokens.sol`). These findings are not an immediate security concern, however showcase proper-

[2] The bit is technically considered for the case `parentCollectionId=0`, which is treated as an exception by the code. This does not influence the security of the contract since (a) there are no curve points with $x = 0$ or $y = 0$ except the point at infinity, encoded as $(0, 0)$, which however is not a solution for the elliptic curve equation, and (b) no bitstring can be mapped to the point at infinity.

[3] Sanitizing the first 254 bits of `x2` to represent a number smaller than or equal to P is however less important, since evaluating the precompiled contract `0x06` on input of points with x coordinate larger than or equal to P yields failure.

ties of the conditional tokens which might not be intuitive nor desirable. With the right calls to the contract, anybody could:

- Tie a token multiple times for the same condition, even using contradictory outcomes.
- Generate tokens that are not tied to any condition (*zero tokens*).
- Tie a token a negative amount of times.

This section describes how to trigger these behaviors.

A collection ID can approximately be described as:

$$H(\texttt{conditionId1}, \texttt{outcome1}) + H(\texttt{conditionId2}, \texttt{outcome2}) + \ldots + H(\texttt{conditionIdN}, \texttt{outcomeN}),$$

where the sum is the sum of elliptic curve points in `alt_bn128` and the pair condition/outcomes are these to which the conditional token is tied.

To tie a token two times under the same condition (in this example with two possible outcomes), it suffices to run:

`splitPosition(collateralToken, 0x0, conditionId, [0b01, 0b10], 1)`

`splitPosition(collateralToken, H(conditionId, 0b01), conditionId, [0b01, 0b10], 1)`

At the end of the execution, three tokens have been created with collection IDs $H(\texttt{conditionId}, \texttt{0b10})$, $H(\texttt{conditionId}, \texttt{0b01}) + H(\texttt{conditionId}, \texttt{0b01})$, and $H(\texttt{conditionId}, \texttt{0b01}) + H(\texttt{conditionId}, \texttt{0b10})$.

A negative tie can be obtained only for a resolved condition. Consider a condition with three possible outcomes that has been resolved so that the only redeemable outcome is `0b001`. The following calls create a token with a negative tie starting from one unit of the token with collection ID `token001` = $H(\texttt{conditionId}, \texttt{0b001})$. A key observation is that points of an elliptic curves form a group, allowing to efficiently compute the additive inverse of an element and thus subtract elements.

`splitPosition(collateralToken, 0x0, conditionId, [0b001, 0b110], 1)`

`redeemPosition(collateralToken,` $H(\texttt{conditionId}, \texttt{0b001}) - H(\texttt{conditionId}, \texttt{0b011})$`, conditionId, [0b011])`

Note that the collection ID of the token we have just redeemed has now a "negative tie" to the chosen condition ID with outcome `0b011`. The zero token can be obtained from the previous token as follows:

`redeemPosition(collateralToken,` $-H(\texttt{conditionId}, \texttt{0b011})$`, conditionId, [0b001])`

`splitPosition(collateralToken,` $-H(\texttt{conditionId}, \texttt{0b011})$`, conditionId, [0b100, 0b011], 1)`

Interestingly, the zero token is not tied to `conditionId` and can be used with any other condition ID that uses the same collateral token.

References

[1] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 319–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. `https://www.cryptojedi.org/papers/pfcpo.pdf`.

[2] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Laih, editor, *Advances in Cryptology - ASIACRYPT 2003*, pages 188–207, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. `https://people.csail.mit.edu/devadas/pubs/mhashes.pdf`.

[3] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Berlin, Heidelberg, 2003.

[4] Thomas Icart. How to hash into elliptic curves. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 303–316, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. `https://eprint.iacr.org/2009/226.pdf`.

[5] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F Aranha. Elliptic curve multiset hash. *The Computer Journal*, 60(4):476–490, 2016. https://arxiv.org/pdf/1601.06502.

[6] Christian Reitwiessner. EIP 196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. *Ethereum Improvement Proposals*, 02 2017. https://eips.ethereum.org/EIPS/eip-196.

[7] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 487–506, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://eprint.iacr.org/2011/339.pdf.

[8] Mark F Schilling. The longest run of heads. *The College Mathematics Journal*, 21(3):196–207, 1990. https://www.maa.org/sites/default/files/images/images/upload_library/22/Polya/07468342.di020742.02p0021g.pdf.

[9] Mehdi Tibouchi. A note on hasing to bn curves. *SCIS. IEICE*, 40, 2012. http://www.normalesup.org/~tibouchi/papers/bnhash-scis.pdf.

A Computing square roots in finite fields

Let $\mathbb{F}$ be a finite field of prime order P such that $P = 3 \mod 4$.

Let X be in $\mathbb{F}$, and assume it has a square root in $\mathbb{F}$ (that is, there exists $Y \in \mathbb{F}$ such that $Y^2 = X$). We show that the element $Y \coloneqq X^{\frac{P+1}{4}}$ is a square root of X.

Note that $\frac{P+1}{4}$ is an integer, since $P = 3 \mod 4$. Thanks to Fermat's little theorem, we know that $X^P = X$ for every element $X \in \mathbb{F}$. In particular, $X^{P+1} = X^2$. Since $P+1$ is divisible by 2, we can rewrite the previous expression as $X^{\frac{P+1}{2}} = X$. Recalling the definition of Y, the previous expression is equivalent to $Y^2 = X$.

B Heuristic worst-case rejection run from bitstrings to group

By Hasse's theorem [3], there are approximately as many points in the curve as there are points in the base field $\mathbb{F}$ (as well as points in the subgroup used by alt_bn128, since the cofactor is 1 For every elliptic curve point (x, y), there is exactly another different point (namely $(x, -y)$) with the same x coordinate (there are no points with $y = 0$). Therefore, picking a random x gives a probability of failing to find a point of about $\frac{1}{2}$. If we assume heuristically that the x coordinates of the curve points are uniformly distributed in the base field, the expected worst case length of a sequence of rejected coordinates corresponds to measuring the longest sequence of consecutive heads when tossing $|\mathbb{F}| \approx 2^{254}$ fair coins. Schilling [8] showed that tossing n independent coins in a row gives an expected longest sequence of heads of length approximately $\log(n)$.[4]

C Security of proposed map from bitstrings to group

First, we describe the framework that we use to construct a multiset hash function H from multisets with elements in a set A to some additive group $\mathbb{G}$. Any multiset hash function is built starting from a standard hash function $\hat{H} \colon A \to \mathbb{G}$ by defining:

$$H(\{a_1^{m_1}, \ldots, a_k^{m_k}\}) = m_1 \cdot \hat{H}(a_1) + \ldots + m_k \cdot \hat{H}(a_k).$$

[4] If the change proposed in Comment 1 is carried out, the worst case number of failures is still about 254. The heuristic assumption on the distribution of points in the base field is not necessary anymore, and the goal becomes finding the maximum length of $|\mathbb{F}|$ independent geometric random variables, again discussed by Schilling [8].

This framework is described as `MSet-Mu-Hash` by Clarke et al. [2], who prove the multiset hash function collision resistant as long as $\hat{H}$ is modeled as a random oracle and the discrete logarithm problem is hard in $\mathbb{G}$ [2, Theorem 2].

In our setting $\mathbb{G} =$ `alt_bn128` [1], where the discrete logarithm problem is assumed to be hard. Recall that `alt_bn128` is defined over a finite field of size P.

It remains to choose an appropriate hash function. The hash function C we propose is built from the function $F\colon A \to \{0,1\}^{256}$, which is modeled as a random oracle. In practice, F can be implemented using `keccak256`. With the symbol / we represent integer division and with `ycoord`(x) a deterministic maps that, given a valid x coordinate of the elliptic curve, computes a y such that $(x, y) \in \mathbb{G}$.

```
counter = 0
repeat:
  h = F(bitstring, counter)
  b = most significant bit of h
  x = remaining 255 bit of h
  counter = counter + 1
until (x < P · (2^255/P)) and (there exists y such that (x, y) ∈ 𝔾)
x = x % P
if (b = 0):  y = ycoord(x); else:  y = P − ycoord(x)
return (x, y)
```

We want to prove that the hash function C described by the previous lines of code can be used in place of the random oracle $\hat{H}$ to build a secure multiset hash function H. To this end, it suffices to show that C is indifferentiable from a random oracle $\hat{H}$ with values in $\mathbb{G}$, assuming F to be a random oracle. Then this choice of $\hat{H}$ can be used to replace a random oracle, since the security game has a single stage [7].

We prove finally that this construction is indifferentiable from a random oracle. This construction is almost equivalent to the "try-and-increment" map, known to be indifferentiable to a random oracle with values in $\mathbb{G}$ [9]. The only difference is that, in their setting, the starting random oracle maps directly to the finite field, while here we need to map bitstrings to the finite fields. To show indifferentiability, we prove that there exists a simulator S that emulates the random oracle F when given access to $\hat{H}$. For a call $S($`bitstring`$, c)$, the simulator is defined as follows (all bitstrings start as not initialized):

```
if bitstring is not initialized:
  counter = 0
  (b, x) = uniform value in {0,1} × {0,1}^255
  while (x ≥ P · (2^255/P)) or (there exists no y such that (x, y) ∈ 𝔾):
    define S(bitstring, counter) = (b, x)
    (b, x) = uniform value in {0,1} × {0,1}^255
    counter = counter + 1
  (x, y) = Ĥ(bitstring)
  if (y = ycoord(x)):  b = 0; else:  b = 1
  q = uniform value between 0 and 2^255/P − 1
  x = x + q · P
  define S(bitstring, counter) = (b, x)
  mark bitstring as initialized
if S(bitstring, c) is not defined:
  define S(bitstring, c) = uniform value in {0,1} × {0,1}^255
return S(bitstring, c)
```

It can be immediately seen that the distribution of the pair (F, C) is the same as that of $(S, \hat{H})$. This shows that the construction C is indifferentiable from a random oracle $\hat{H}$.[5]

[5]The simulator S needs to be efficient, hence there need to be a cutoff for the number of instructions. The contribution of the cutoff is too small to matter. A distinguisher might be able to call out the simulation after the cutoff, however the probability to exceed the cutoff size decrease exponentially with the height of the cutoff, as it can be inferred from appendix C: If the distinguisher was able to make 2^{256} queries, the expected maximum length of a loop the adversary could cause is only 256. By

setting the cutoff at 1000, the distinguishing probability becomes irrelevant for any practical number of queries.