Groth16 zk-SNARK Proof Verification on Freeton : Three Use Cases

OCamlPro - Fabrice Le Fessant*and Thomas Sibut-Pinote†

https://github.com/OCamlPro/devex-18-zk-contest

1 Introduction

1.1 Description of the submission

The present document is an official submission to the 18th contest[1] of the DevEx governance: *Groth16 zk-SNARK Proof Verification Use Cases*. Although we accepted earlier to be part of the jury, we understand that by submitting this document we waive our right to vote in the present contest.

In this submission, we propose three different use cases for Groth16 zk-SNARK proofs on the FreeTON blockchain:

- (Section 2) Sudoku: zk-SNARKs are used to prove that the user has a solution to a given Sudoku problem, without providing the solutions to other users ;
- (Section 3)Project Euler: zk-SNARKs are used to prove that the user has found the solution to a math problem from the famous https://projecteuler.net website;
- (Section 4)Pin-code reset of forgotten keys: zk-SNARKs are used to replace a pubkey when the user has lost his secret key.

We then show the contributions that we made for the Free TON community while working on this contest (Section 5). Finally, we discuss the three paradigms of using zk-SNARKs that our examples showcase (Section 6).

1.2 General remarks

ft: Throughout this project, we make heavy use of our in-house freeton wallet and client `ft`. We have taken extra care to make its use very simple, using docker images and intuitive, well-documented features. We have specially adapted this tool to be able to create local sandboxes using the Nil Foundation's fork of the TON OS, as well as to interact with the Nil foundation server. Please check out https://hub.docker.com/r/ocamlpro/

*Telegram: @fabrice_dune

†Telegram: @ThomasSibutPinote

[1] https://devex.gov.freeton.org/proposal?proposalAddress=0%3Ae6b65075478e7d412fdb0870452f30dfa8bf51272e28a3167abc5c5df6fd051d

ft. The documentation is available both in the command line through `ft --help` and on the website https://ocamlpro.github.io/freeton_wallet/sphinx/.

Code: All our code comes with a README, standard in-code documentation and tests. This document is provided as an overview of our use cases for zk-SNARKs on the Freeton blockchain but you can also directly dive into the code. All our code is under the GPL license and is hosted at https://github.com/OCamlPro/devex-18-zk-contest.

Compilation time: The long compilation time of the repository provided was a major obstacle in our progression. We would suggest trying to make it shorter for future contests.

Marshalling and memory corruption: Because of a memory corruption error in the marshalling function (in the code provided for the contests) of the verification key, we were unable to do a full 9 by 9 Sudoku, restricting ourselves to 4 by 4. All our code is generic (it is generated from an OCaml program which takes as a parameter the value n such that the Sudoku is n^2 by n^2). We were told that after the contest, a more robust marshalling library would be provided. In any case we would be glad to discuss this issue with the Nil foundation team to give users the full Sudoku experience! In the meantime, a 4 by 4 Sudoku is sufficient to get a sense of the dynamics of zk-SNARK-based Sudoku.

Keys on contracts: In the Euler and Pincode smart contracts, we provided, along with the verification key as needed by the `vergroth16` instruction, the proving key (in a zipped format) on the smart contract. This enables users to easily find it in order to generate their proofs. In case the contract deployer should decide to keep the possibility of changing the verification and proving keys of the problem(s), this also makes it easiest to follow changes before submitting a solution. Unfortunately, the proving key for Sudoku is an order of magnitude bigger than for Euler and Pincode (even zipped), and we decided not to include it.

Separate submissions: After deliberation, we decided to submit only one document rather than three separate submissions, in a spirit of coherence. We leave it to the appreciation of the jury whether this work deserves to be counted as several submissions.

Thanks: We would like to thank @noam for his efforts and sportsmanship during the contest, and to the Nil foundation team for their answers to our various questions.

2 A toy example: the Sudoku grid

Let's start with our first toy use case. Suppose you want to set up a Sudoku problem for your students to solve, so that upon completion they receive some token. The issue is that once any student has solved it, the solution sits on the blockchain for all to see. All the other students can cheat by copying it, instead of doing the work by themselves, which defeats the whole purpose.

1								6
		6		2		7		
7	8	9	4	5	6	1	2	3
			8		7			4
				3				
	9				4	2		1
3	1	2	9	7			4	
6	4			1	2		7	8
9	7	8						

Figure 1: A Sudoku grid

As mentioned in Section 1.2, we had to restrict ourselves to a 4 by 4 Sudoku grid, even though our code generator is generic in the grid size. This is because the marshalling function (in the codep rovided for the contest) on the 9 by 9 code produced a memory corruption error.

2.1 Mathematical encoding

Note: This section may be skipped upon a first reading without loss of information.

Let n^2 be the size of a Sudoku grid (in our case, $n = 2$ and $n^2 = 4$ is the length of a side of the Sudoku square).In the following, the variables $(x_{i,j})_{0 \le i,j < n^2}$ denote the (secret) values of the solution of a given instance of a Sudoku. The variables $(f_{i,j})_{0 \le i,j < n^2}$ denote the instance so that

$$f_{i,j} = \begin{cases} 0 & \text{if square } i,j \text{ is not fixed by the Sudoku instance} \\ \text{The value in square } i,j & \text{otherwise} \end{cases} \tag{1}$$

The (well-known) constraints of a Sudoku are encoded by the following equations:

$$\prod_{k=1}^{n^2} (x_{i,j} - k) = 0 \text{ for all } 0 \le i, j < n^2$$

and

$$f_{i,j} \left(x_{i,j} - f_{i,j} \right) = 0 \text{ for all } 0 \le i, j < n^2$$

and, for all S set of indices representing a row, a column or a box as per the rules of Sudoku:

$$\prod_{(i,j) \in S} x_{i,j} = \prod_{k=1}^{n^2} k = n^2!$$

and

$$\sum_{(i,j)\in S} x_{i,j} = \sum_{k=1}^{n^2} k = \frac{n \cdot (n^2+1)}{2} \, .$$

The first equations guarantee that the only permitted values for $x_{i,j}$ are the integers from 1 through n^2, while the second force

$$\forall 0 \leq i,j < n^2, f_{i,j} \neq 0 \implies x_{i,j} = f_{i,j} .$$

The last equation, by forcing the product of n^2 integers between 1 and n^2 to be equal to $\left(n^2\right)!$, forces every row, column and box to contain one and only one instance of the integers 1 through 9.

2.2 Components

This use case includes two different components:

The cli: it is a C++ program linked to the Blueprint zk-SNARKs library. It can be used in two different ways:

- To create the original proving and verification keys. This step is independent of any given Sudoku instance and should only be done once.

  ```
  ./sudoku-client --sudoku-generate-keys
  ```

- To generate a proof given an instance and a solution.

  ```
  ./sudoku-client --sudoku-generate-proof --instance
    instance.in --solution solution.in
  ```

  The instance is given as a text file such as

  ```
  0000
  0000
  0000
  0001
  ```

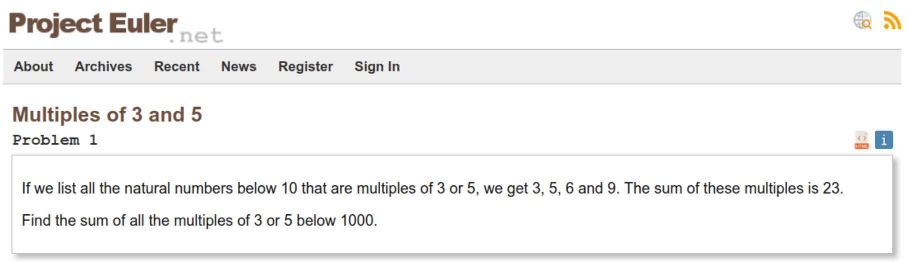
  where 0s denote free values. The solution should be a similar text files with no zeroes, such as

  ```
  1234
  3412
  2143
  4321
  ```

  Note that if your solution is not correct, this step will fail.

The Sudoku smart contract: The only contract for this use case is the `Sudoku` contract. Only the deployer's public key is allowed to submit new Sudoku instances. Its constructor defines an initial instance and the verification key (in this case the proving key is not included in the smart contract due to its size). The main function `submit` takes as input a proof and returns a boolean stating its correctness. The local function `pi_from_instance` encodes the primary input given the current instance, but does not store it anywhere. The `submit_instance` allows the owner to add a new challenge at any time.

3 Project Euler



Multiples of 3 and 5

Problem 1

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

Figure 2: The first Project Euler problem

The previous example is somewhat limited by the fact that Sudokus are easily solvable problems, either by hand or with a computer; they are not the most convincing use case of (at least the zero-knowledge aspect of) zk-SNARKs[2].

Let's keep the idea of a decentralized classroom. the famous website Project Euler[3] offers math and programming problems of various difficulty to its users. It uses captchas to prevent brute force attempts at guessing the answer to a problem. The answers to problems are stored statically on its servers and the answers are checked against them. Can we make Project Euler decentralized? Using zk-SNARKs, we can prevent users from stealing others' solutions **and** from brute-forcing them, due to the cost of generating proofs.

3.1 Components

This use case includes two different components:

The C++ client The 'euler-client' program can be used in 2 ways:

1. `./euler-client prepare PROBLEM NONCE SOLUTION`
   , where `PROBLEM` is a problem number, `NONCE` is an arbitrary number different for every problem, and `SOLUTION` is the numeric solution, will generate a proving key and a verification key for the problem and its solution. These files can be published for users to propose submissions and to verify them.

2. `./euler-client prove PROBLEM NONCE SUBMISSION PUBKEY`

[2]Although they hint at a possible use for zk-SNARKs: checking a result without implementing its full logic in Solidity may save precious storage and computations on the blockchain.

[3]`https://projecteuler.net/`

, where `SUBMISSION` is the numeric solution proposed by the user and `PUBKEY` is his own public key (to prevent other users from submitting the same solution). The program generates a proof to be then submitted online.

Solidity Smart Contracts The Solidity smart contracts are located in the 'contracts/' directory. They can be built with 'make' if 'ft' (freeton-wallet) is installed.

The following smart contracts are defined:

1. `EulerRoot` : it's the central contract of a set of Euler problems. The `EulerRoot` contracts is owned by the organizer, who is the only one able to publish new Euler problems. Its most important functions are `new_problem()` to deploy a new problem contract, `new_user()` to deploy a user contract (anybody can call it) and `submit()` to submit a solution for a problem. `problem_address()` and `user_address()` are useful to get the contract address of a problem or a user. An event is emitted everytime a problem is solved by a user.
2. `EulerProblem`: it's the contract corresponding to a given Project Euler problem. The contract contains the verification key of the problem, so that it can check the proofs submitted by users. Since proofs are attached to users' pubkeys, other users cannot re-use proofs submitted by other users. Each `EulerProblem` contract contains the top 10 of the first users to submit a correct solution. An event is also emitted everytime the problem is solved by a user. The contract also contains the description of the problem (title, description, url) and the information needed to submit solutions (compressed proving key and nonce).
3. `EulerUser`: it's the contract corresponding to the pubkey of a user. The user can provide a name using the `set_name()` function. The `has_solved()` function is called by the `EulerRoot` contract when the user has solved a new problem. The function stores the problem and the time of solution in the contract.

For internal messages between these contracts, they either check that messages are coming from the unique `EulerRoot` contract, or that they come from pre-computed addresses à la TIP-3.

3.2 Brute-force protection

This small section details an interesting aspect we had to develop to prevent bruteforce attacks.

Project Euler is a gentlemen's competition. For example, competitors are supposed to have found the answer using a program running at most in one minute, but no mechanism enforces this. Still, some protections are in place, partly to enforce the spirit of the competition, and partly because of past abuse. For instance, a captcha system prevents one from attempting many solutions in a row. We decided to emulate the latter mechanism. If we simply gave a proving key to try and find the actual solution to the problem, users might be tempted to try to build a proof using every single integer until it works[4]. Hence, the value they must find is actually

$$\text{SHA256}^{1\,000\,000}\,(problem\_number|solution|nonce)$$

where the exponent denotes repeated application of the SHA256 hashing function, and '|' denotes the concatenation of strings. Note that this does not protect against replay attacks, which is why we also included a form of authentication described in Section 4. This repeated application takes about 20 seconds on our laptops, which seems enough to discourage brute-force by all but the most determined cheaters.

To prevent users from easily brute-forcing the numeric solutions, we added a simple proof of work of about 20 seconds. The proof of work is based on 1,000,000 iterations of sha256 over the concatenation of the problem number, the nonce and the solution. Since a random nonce included in the hash, the proof-of-work cannot be done before the new problem has been submitted and the corresponding new nonce published.

4 Pin code reset for forgotten keys

In this use case, we use zk-SNARKs to allow users to replace their public keys in other contracts, for example multisig wallets. The typical use is when a user has lost the associated secret key, and is unable to sign/confirm any new transaction. With our approach, the user first creates a recovery contract, where a passphrase is associated to his public key. If he needs to change his public key, he can use zk-SNARKs to associate a new public key in his recovery contract, and tell other contracts to safely replace his pubkey by the new one.

Compared to other approaches, this use can provide several improvements:

- The user does not disclose his passphrase. Most other approaches (based on hashes for example) give an opportunity for man-in-the-middle attacks;
- As the user does not disclose his passphrase, the pubkey can be replaced several times;

[4]We have noticed that although generating a correct proof may take time, an incorrect input is usually rejected in less than one second.

- The user does not need to create the new public key ahead of time. Instead, he can create it when he has lost the former one,

  decreasing the likelihood of losing the associated secret key also.

4.1 Technical solution

This use case includes 3 different components:

PinCode Client: it is a C++ program linked to the Blueprint zk-SNARKs library. It can be used in two different ways:

- To create the initial circuit using the passphrase. The client should be called as:

  ```
  pincode-client prepare "my pass phrase"
  ```

  This will generate a file `verifkey.hex` that can be used as an argument when deploying the `PubkeyRecovery` smart contract, and a file `provkey.hex` that can be used to create new witnesses;

- To create a witness that the passphrase is known when providing a new public key. Because the witness contains the new public key, an attacker cannot intercept the message and replace it. The client should be called as:

  ```
  pincode-client prove "my pass phrase" "PUBKEY_AS_HEX_NUMBER"
  ```

  The command expects to find the file `provkey.hex` and generates a file `proof.hex` that should be submitted to the `PubkeyRecovery` smart contract together with the new public key.

PubkeyRecovery smart contract: This smart contract is a contract à la TIP-3 contract, using the initial public key to verify its address. This contract is very simple. It contains:

- A constructor to set the circuit that will be used to verify that the passphrase is known when it is used;
- An external function `SetFromPincode(bytes proof, uint256 pubkey)` to define the new public key, while providing a proof of knowledge of the passphrase;
- An external function `RecoverPubkey(RecoverablePubkey addr)` to call a recoverable contract to update the corresponding pubkey.

Abstract RecoverablePubkey contract: This is an abstract contract, from which other contracts should inherit to be able to benefit from the `PubkeyRecovery` mechanism. The contract provides two main functions:

- An external function `SetPubkeyRecoveryCode(TvmCellcode)` to define the code of the `PubkeyRecovery` smart contract. Because the code hash is known (inlined in the code), anybody can call this function with the correct code.

- An external function `RecoverPubkey(uint256 oldkey,uint256 newkey)` that can only be called by a `PubkeyRecovery` smart contract. The function verifies that the address belongs to such a contract before calling a function `recover\_pubkey(oldkey, newkey)`, that the inheriting contract should define to specify the business logic that should happen when a pubkey is replaced.

Modified Multisig Wallet: The `RecoverableMultisigWallet` contract is the standard multisig wallet of Surf, updated for version 0.40 of Solidity, and modified to inherit from the `RecoverablePubkey` contract. It defines the business logic to replace a custodian key. For that, it first checks that the old key belongs to an existing custodian of the contract, that the new key does not belong to an existing custodian, and then replace the old key by the new one, associated to the same index (for bitfields).

5 Contributions to the Free TON community

In the course of participating to this contest, we were led to refine our in-house tools in order to more easily manipulate the Nil Foundation forks of the Ton Virtual Machine (TVM), the TON Solidity compiler and the TVM linker.

Our first contribution is a public Docker image containing the TONOS SE compiled with zk-SNARKs support. The image is published as `ocamlpro/nil-local-node` and is built from https://github.com/NilFoundation/tonos-se.

Running it is simple:

```
docker run -d --name local-node -e USER_AGREEMENT=yes -p80:80 ocamlpro/nil-local-node
```

Another way to use it is through our development tool `ft`, a Free TON Wallet designed for developers: https://github.com/OCamlPro/freeton_wallet

For this contest, a set of improvements has been contributed to `ft` to ease using it.

In particular:

- The interface has been improved to provide multiple levels of subcommands, making them easier to understand and to use;
- A new argument `-image` is available with `ft switch create` to specify the Docker image to use. This new argument is specially useful for our Docker image for NilFoundation TONOS SE.

As a consequence, a zk-SNARK-ready sandbox (local network) can be installed by simply running:

```
ft switch create sandbox --image ocamlpro/nil-local-node
```

Now all commands from the `ft` documentation will work, in particular it will be easy to create accounts, deploy contracts, and call them as seen in https://ocamlpro.github.io/freeton_wallet/sphinx/use-cases.html.

The Docker image of `ft` has been updated to use NilFoundation tools (`solc`, `tonos-cli`, `tvm_linker`), it is the easiest way to use `ft` if you don't want to build it. See the installation instruction at (https://ocamlpro.github.io/freeton_wallet/sphinx/install.html#using-docker).

6 Thoughts on three paradigms of zk-SNARKs on blockchains

We wish to conclude with some broader considerations. This contest has given us the opportunity to think harder about the categories of uses of zk-SNARKs on blockchains:

1. Statically checking that someone has some information at their disposal(Euler case)
2. Encoding a possibly complex computation in a circuit and challenging a user with an instance of that computation (Sudoku case)
3. Using the zk-SNARK as an extension of the blockchain protocol itself (pincode, anonymous transactions, voting protocols)

Of course, the boundaries between these categories may be blurred, but still they are meaningful. The point of the first category is that the data whose existence we are verifying is inert, static, it was chosen by the verifier according to some human meaning which is completely meaningless to the contract or the blockchain. In the second category, the value whose existence we are verifying has a computational (i.e. mathematical, see Section 2.1) meaning, and though this computational meaning[5] is not present on the blockchain itself, it is implicitly present in the verification key. Finally, the third category consists in the verification of data which has blockchain-protocol-level computational meaning (such as a public key). The privacy aspect of zk-SNARKs, which is of course the whole point, had initially hidden these distinctions from us.

In category 1, the main advantage of zk-SNARKs is to not spoil the fun for other competitors. One must protect oneself from replay attacks (re-using the same zero-knowledge proof as some other competitor in order to pretend one has solved the problem) and brute-forcing the answer.

In category 2, the point is to guarantee some mathematical properties of a wide range of submitted solutions to a given problem of a known computational nature, when not all of possible instances may be computed in advance. The *zero-knowledge* part may or may not be important, but the *succinct* part may well save some gas for cost-heavy verifications.

In category 3, the zero-knowledge part plays a crucial part in preventing man-in-the-middle attacks for security-critical operations such as changing a lost public key in a wallet, or sending an anonymous amount of tokens to an anonymous recipient, or voting without revealing one's ballot.

These three categories may be combined in several ways depending on the purpose.

[5] Of course, the answers to the Project Euler problems do have computational meaning ultimately, but no program is provided to encode the problem or check the solution. This computational meaning is inferred from the english description of the problem by the problem solver.