SCIS 2020 2020 Symposium on
Cryptography and Information Security
Kochi, Japan, Jan. 28 – 31, 2020
The Institute of Electronics,
Information and Communication Engineers

Smart contract with secret parameters

Marin Thiercelin [*†] Chen-Mou Cheng [*] Atsuko Miyaji [*] Serge Vaudenay [†]

Abstract: By design, smart contracts data and computations are public to all participants. In this paper we study how to create smart contracts with parameters that need to stay secret. We propose a way to keep some of the parameters off-chain, while guaranteeing correctness of the computation, using a combination of a commitment scheme and a zero-knowledge proof system. We describe an implementation of our construction, based on ethereum smart contracts, and zk-SNARKS. We provide a small example and do a cost analysis of our approach.

Keywords: smart-contract, privacy, decentralized, zero-knowledge, ethereum

1 Introduction

The aim of smart contract systems is to secure relationships, over computer networks, without the needs of legal third parties. This idea has been brought to reality [1] by achievements in the fields of distributed systems and cryptography. Existing smart contract systems use a decentralized protocol, which requires all the information and computations involved in the contract to be public. This particularity provides a strong accountability, as everything is logged and verified by the decentralized system. However the public nature of smart contracts can also be a drawback, as many applications require privacy of some of the information involved in the contract. In this work, we want to look at constructions that make it possible to use smart contracts where some parameters of the contracts need to stay secret, while preserving the correctness of the contracts.

2 Background

2.1 Notations

We present notations used in this paper.

1. We write the security parameter as λ
2. We say a function ϵ is negligible (in λ), iff $\forall c \in \mathbb{N}^*, |\epsilon(\lambda)| = \mathcal{O}(\frac{1}{\lambda^c})$
3. We say an algorithm is ppt, when it is a randomized algorithm with polynomial asymptotic complexity in λ
4. We write $x \leftarrow_\$ X$, when we sample x uniformly in the set X. We write $y \leftarrow A(x)$ when y is the output of a randomized algorithm A on input x. We write $y := A(x)$ when y is the output of a deterministic algorithm A on input x
5. We write $a := \langle b|c \rangle$ to define a as the concatenation of b and c, and $\langle b|c \rangle := a$ when we decompose a into b and c
6. We write the states of a smart contract as $\boldsymbol{s_i}$. We write $\boldsymbol{s_i}[x]$ when we want to access a particular variable x in the state $\boldsymbol{s_i}$
7. We say a computation is executed onchain when it is executed by all the nodes of the bockchain. We say that it is executed off-chain when it is executed locally by one peer

2.2 Blockchain

A blockchain is a decentralized protocol, where the decision power is shared by all the parties involved (also called nodes, or peers). A blockchain is used by peers to settle on a shared, sequential state history. For a fixed period of time, the nodes broadcast a set of changes, called transactions, that they want to see in the next state. At the end of the period, using a consensus algorithm, the nodes decide on a set of state changes, called a block, and add it to their state history (which forms a chain of blocks, hence the name). Blockchains use cryptographic and distributed systems to provide some useful guarantees: the blockchain state cannot be corrupted, and it's not possible to change the state history retroactively. Among other things, a blockchain can be used to power a digital currency [2]. Peers sending transactions are identified using public/private key pairs and digital signatures, to make sure only authorized transactions are accepted by the blockchain. Another important property of blockchain is that a new block is computed at regular time intervals, thus the block count can be used as a clock by applications.

2.3 Smart contract

As discussed in introduction, a smart contract is to be used as a contract, in an online and decentralized setting. It needs to secure relationships, without the

[*] Miyaji Laboratory, Osaka University
[†] LASEC, EPFL

use of trusted authorities and legal systems. In practice, smart contract systems are implemented as a programming language on top of a blockchain protocol. The smart contract is defined as a contract state and a piece of computer code that is included (we say that it is deployed) in the blockchain state. When a transaction is sent to the contract, the code is executed and produces a new contract state, which is included in the blockchain. More interestingly, smart contracts can be programmed to hold and transfer digital assets.

2.4 Commitment scheme

A commitment scheme is a cryptographic primitive, which can be used in a protocol to force a party to commit to a secret value before continuing the protocol.

Definition 1 (Commitment scheme). *A commitment scheme consists of three efficient algorithms :*

- *Setup: randomized algorithm, takes in the security parameter and output system parameters σ*
- *Commit: randomized algorithm, given a value $v \in V$ returns a commitment c and a key k*
- *Open: given a commitment c and key k outputs a value in V or $\perp$ (meaning an incorrect opening)*

That satisfy the following:

- *Correctness: Let $\sigma \leftarrow Setup(1^\lambda)$, $\forall v \in V$, if $(c, k) \leftarrow Commit_\sigma(v)$,then $Open_\sigma(c, k) = v$*
- *Hiding: Any ppt adversary, choosing $v_0, v_1 \in V$, and given a commitment c of value v_b, with b a random bit, cannot output a bit $b' = b$, with probability significantly larger than $\frac{1}{2}$*
- *Biding: Any ppt adversary can output (c, k, k') such that $\perp \neq Open_\sigma(c, k) \neq Open_\sigma(c, k') \neq \perp$, only with negligible probability*

2.5 Proof system

A proving scheme is used in a protocol with 2 parties, the prover and the verifier, where the prover wants to convice the verifier that a statement is true.

Definition 2 (Non-interactive proof-of-knowledge). *For an efficient boolean function $R(a, w)$, and associated language $L = \{a|\exists w, R(a, w) = 1\}$, if $R(a, w) = 1$, we say that w is a witness of the statement $a \in L$. For any such language L, a non-interactive proof-of-knowledge system consists of three efficient algorithms:*

- *Setup: randomized algorithm, takes in the security parameter and output system parameters η*
- *Prove: randomized algorithm, given a statement a and witness w returns a proof π*
- *Verify: given a statement a and proof π, returns a bit*

That satisfy the following:

- *Completeness: $\forall a \in L$, if $\eta \leftarrow Setup(1^\lambda)$, $R(a, w) = 1$ and $\pi \leftarrow Prove_\eta(a, w)$, then $Verify_\eta(a, \pi) = 1$*
- *Soundness: $\forall a \notin L$, Any ppt adversary produces a proof π', such that $Verify_\eta(a, \pi') = 1$, only with negligible probability*

Zero-knowledge proof In our construction, we will require an additional property : the proof should be zero-knowledge. Informally, the zero-knowledge property means that any efficient adversary, given a correct proof π, only learns that $a \in L$ is true, and specifically, doesn't learn anything about the witness w.

2.6 Related work

In 2016, the Hawk team proposed a construction for privacy-preserving smart contracts [3]. Their aim was to achieve transaction privacy, where the transaction data (sender, value) is not disclosed. They use a special party, the manager, which, is trusted for privacy. Correctness is enforced using zero-knowledge proofs. Another line of work has been using zero-knowledge proofs to introduce privacy in blockchain transactions; the zerocash protocol [4] extends the bitcoin protocol, with hidden transactions, where the amount, sender and receiver are not shared with anyone else. In the Zexe paper [5], an extension of the zerocash protocol is proposed, which allows programmable coins. Zexe successfully realizes a primitive they named *decentralized private computation*, that can be useful in applications such as private digital currency exchanges.

2.7 Our contribution

We propose a construction to use smart contracts, for applications with private parameters. Our solution works directly with existing smart contract systems. We propose a library that implements our construction with ethereum smart contracts. Our paper is organized as follows, in Section 3 we formally define the problem we want to solve, and give a small example to illustrate. In Section 4 we propose a construction for a smart contract with private parameters, that satisfies the requirements of Section 3. In Section 5 we described an implementation of our construction using ethereum smart-contracts. We analyze the performance of our implementation in Section 6. In Section 7, we discuss the drawbacks of our construction and evoke some possible directions to address them.

3 Problem definition

In this paper, we study applications defined by an initial state $\boldsymbol{s_0}$, and a transition function $IdealTransition$. A special party, the owner, chooses a secret value $secret$, which is used as a parameter by the transition function. When other parties, the users, input the application with some data $data$, the next state of the application is computed as $\boldsymbol{s_{i+1}} = IdealTransition_{secret}(\boldsymbol{s_i}, data)$.

As we said in the introduction, these applications can't be implemented using smart contracts alone, as their public nature would leak the value *secret*. In this paper, we looked at a relax problem, where we allow an implementation to enter an intermediate state $\boldsymbol{s'_i}$, and eventually reach the correct state $\boldsymbol{s_{i+1}}$. We can break down *IdealTransition* into subroutines f, g, h, k, as in algorithm 1.

Algorithm 1 $IdealTransition_{secret}(\boldsymbol{s_i}, data)$

```
$\boldsymbol{s'_i} := k(\boldsymbol{s_i}, data)$
$z := h(\boldsymbol{s'_i})$
$y := f(secret, z)$
$returng(\boldsymbol{s'_i}, y)$
```

Subroutines f, g, h, k have distinct roles:

- k includes all the computations independent of *secret*, and produces an intermediate contract state $\boldsymbol{s'_i}$
- h extract the input in a format expected by f
- f is a deterministic function of the value *secret* and the value z produced by h
- g links everything and computes the next state

Now we can formally define the security we want to achieve:

Definition 3. *We say that a protocol P securely realizes $IdealTransition_{secret}$, if it satisfies the following requirements:*

1. *No efficient adversary should gain any knowledge on secret, other than the values $f(secret, h(\boldsymbol{s'_i}))$, with $s'_i := k(\boldsymbol{s_i}, data)$, of each previous state transition*
2. *For any user action with input data at state $\boldsymbol{s_i}$, P eventually gets to a new state $\boldsymbol{s_{i+1}} = IdealTransition_{secret}(\boldsymbol{s_i}, data)$*

Example 1 shows a simple game that could be implemented using this kind of contract.

Example 1 ("Higher or Lower?" game)**.** *Here we have two parties, the owner and the player. The game is parameterized by a set $S \subset \mathbb{N}$, and a number of query t. The owner chooses $s \leftarrow_\$ S$ and let the player make some guesses, and on each guess $g \in S$, returns whether s is a number higher, lower or equal to g. After t guesses, the player makes a final guess, and wins if it is a correct guess. We've described this game in Figure* **??**.

If we can build a smart contract with private parameters that satisfies the requirements, then we can realize the game in the following way. Set the secret parameter *secret*, to the value to guess s. We let the state of the contract hold a counter of queries $cnt = 1$. We define f, g, h, k as in Figure **??**.

This smart contract can act as a "Higher/Lower" oracle, and designate the winner, with a potential reward.

4 Construction

In this section, we introduce a construction for a smart contract with private parameters, that satisfies the requirements of Section 3. Here we describe an incremental development: we give constructions that satisfies the requirements using some assumptions, and we use more realistic assumptions at each step.

4.1 Naive approaches

4.1.1 Regular smart contract

In regular smart contract systems, all parameters (including *secret*) are directly included in the binary of the smart contract. In that case, when a transaction gives input *data* to the contract, the blockchain nodes have all the information necessary to compute the next state $\boldsymbol{s_{i+1}} := g(\boldsymbol{s'_i}, f(secret, h(\boldsymbol{s'_i})))$ with $\boldsymbol{s'_i} = k(\boldsymbol{s_i}, data)$, we say that f is computed on-chain. The incorruptible nature of regular smart contracts automatically satisfies Requirement 2. However, Requirement 1 is satisfied if we make the assumption that no other parties will look at the value of *secret*, which is accessible publicly in the binary of the contract. This assumption is really strong and unrealistic, especially if knowing *secret* can provide some financial gains (e.g win a reward in Example 1).

4.1.2 Off-chain computation

If we wan to protect against malicious users, then the parameter *secret* can't be included directly in the contract. A natural approach is to have the owner keep the value *secret* locally, and publish the smart contract without *secret*. When a user sends a transaction with data *data*, the contract gets to the new state $\boldsymbol{s'_i} := k(\boldsymbol{s_i}, data)$. The owner has to monitor the state changes of the contract, when the smart contract get to a state $\boldsymbol{s'_i}$, the owner computes $z := h(\boldsymbol{s'_i})$ and $y := f(secret, z)$ locally (off-chain), and sends y to the smart contract. The new state is then computed as $\boldsymbol{s_{i+1}} := g(\boldsymbol{s'_i}, y)$. Here Requirement 1 is satisfied by design. However Requirement 2 is satisfied if we assume that the owner is honest. This assumption is arguably less strong, as in many cases the owner will be a company with a reputation and regulations, and will be less tempted to cheat the protocol. However, this assumptions reduces the application to a client-server application, with the smart contract only acting as a middle-man that transfers both messages and assets.

4.2 Our proposal: off-chain and verified

If we want to protect against both malicious users and owners, the smart contract (and the nodes of the underlying block-chain) should be able to verify that the value y returned by the owner, really corresponds to $f(secret, h(\boldsymbol{s'_i}))$. To realize that, we use a commitment scheme and a zero-knowledge proof-of-knowlege system, for the language:

$$L_f = \{a =: \langle z|c|y\rangle|_{\mathrm{Open}_\sigma(c,k)=sk \cap f(sk,z)=y}^{\exists w=:\langle sk|k\rangle,}\}$$

We also need the developer of the contract to define l_{fake} and l_{timeout}, that will be used if the owner doesn't follow the protocol. We assume that the *Setup* algorithm of both the commitment scheme and proof system was run honestly, and the system parameters σ and η are known by all parties.

Initialisation: On deployment, the owner chooses its secret parameter *secret*, computes a commitment $(c, k) \leftarrow \text{Commit}_\sigma(\mathit{secret})$, and includes the commitment c in the code of the smart contract, and keep the key k locally.

Start transition: When a user sends a transaction with data *data*, the contract gets to the new state $\boldsymbol{s_i'} := k(\boldsymbol{s_i}, \mathit{data})$.

Off-chain computation: The owner monitors the state changes of the contract. When the smart contract get to a state $\boldsymbol{s_i'}$, the owner computes $z := h(\boldsymbol{s_i'})$, $y := f(\mathit{secret}, z)$, and $\pi \leftarrow \text{Prove}_\eta(\langle z|c|y\rangle, \langle \mathit{secret}|k\rangle)$, to prove that $\langle z|c|y\rangle \in L_f$.

Finish transition: The owner then sends y and π to the smart contract. The smart contract, computes $z' := h(\boldsymbol{s_i'})$, reconstructs $\langle z'|c|y\rangle$ from the value it stored and the y it received, and computes $b := \text{Verify}_\sigma(\langle z'|c|y\rangle, \pi)$. If $b = 1$ the smart contract accepts y as the value $f(\mathit{secret}, z')$ and the next state is computed as $\boldsymbol{s_{i+1}} = g(\boldsymbol{s_i'}, y)$. If $b = 0$, the smart contract goes to the next state $\boldsymbol{s_{i+1}} := l_{\text{fake}}(\boldsymbol{s_i'})$.

Timeout: If the contract stays too long in state $\boldsymbol{s_i'}$, the user has the possibility to send a "Timeout" transaction, when the contract receives it, it goes to a state $\boldsymbol{s_{i+1}} := l_{\text{timeout}}(\boldsymbol{s_i'})$.

Figure **??** describe the structure of the onchain computation, and Figure **??** describes the protocol as a whole.

Security

Proposition 1. *For any rational, ppt adversary, we can select l_{fake} and $l_{timeout}$, such that our protocol satisfies the requirements in Section 3.*

Proof.
Requirement 1:
In this construction, the only values that are published by the owner is c, y and π. The hiding property of the commitment scheme, guarantees that no information on *secret* is leaked by c, for any ppt adversary. The zero-knowledge property of the proving scheme guarantees that no information on $w := \langle \mathit{secret}|k\rangle$ is leaked, for any ppt adversary. The only information on *secret* that is released is the value $y := f(\mathit{secret}, h(k(\boldsymbol{s_i}, \mathit{data}))$, therefore it satisfies Requirement 1.
Requirement 2
If the owner follows protocol, the completeness property of the proof system, guarantees that the Verify procedure will return 1, and $\boldsymbol{s_{i+1}} := g(\boldsymbol{s_i'}, y)$, it follows that Requirement 2 is satisfied. If the owner doesn't follow protocol, we have to look at several cases.

1. If the owner simply doesn't respond, then it requires the user to send a "Timeout" transaction after some time, and the the contract goes to state $\boldsymbol{s_{i+1}} := l_{\text{timeout}}(\boldsymbol{s_i'})$
2. If the owner sends $y' \neq f(\mathit{secret}, z)$ and π', then there are two cases
   (a) π' is not a valid proof. Then the soundness property of the proof system guarantees that $\text{Verify}_\eta(\langle z|c|y\rangle, \pi) = 0$, and $\boldsymbol{s_{i+1}} := l_{\text{fake}}(\boldsymbol{s_i'})$
   (b) π' is a valid proof. Then the completeness property guarantees that $\text{Verify}_\eta(\langle z|c|y\rangle, \pi) = 1$, and $\boldsymbol{s_{i+1}} := g(\boldsymbol{s_i'}, y')$

In all those cases, the final state $\boldsymbol{s_{i+1}}$ is likely to differ from the value returned by $IdealTransition(\boldsymbol{s_i}, x)$, hence Requirement 2 is not satisfied. To mitigate case 1 and 2a, we can write l_{fake} and l_{timeout} such that they include heavy financial penalties for the owner in the new state, to deter a rational and malicious owner. In the case 2b, if π' is valid, it means that the owner knows a witness $w := \langle \mathit{secret}'|k'\rangle$ for the statement $\langle z|c|y'\rangle \in L_f$. The definition of L_f implies $\text{Open}_\sigma(c, k') = \mathit{secret}' \cap f(\mathit{secret}', z) = y'$. since $y' \neq f(\mathit{secret}, z)$, we have $\mathit{secret}' \neq \mathit{secret}$ (f is deterministic). We also have $\text{Open}_\sigma(c, k) = \mathit{secret}$ from the correctness of the initial commitment. It implies the owner knows (c, k, k') with $\mathit{secret} = \text{Open}_\sigma(c, k) \neq \text{Open}_\sigma(c, k') = \mathit{secret}'$, the binding property of the commitment guarantees that this happens with negligible property for a ppt adversary. □

To wrap up, our construction satisfies the requirements of Section 3, under the assumption that the *Setup* algorithms of the commitment scheme and proof system are executed correctly, for all rational ppt adversaries.

5 Implementation

In Section 4.2, we described a construction to build a smart contract that behave as our Ideal Contract described in 3. We made assumptions on the type of adversary, and we assumed a trusted setup for the commitment and proof system. Since our goal is to have a construction that can be implemented on existing smart contract systems, we have limited the assumptions we made about the capabilities of the contracts. To simplify, we have will restrict our use-cases to applications where the secret parameter *secret* is a single 128-bit unsigned integer, and f is a function taking two 128-bit unsigned integers and returning another 128-bit unsigned integer.

5.1 Tools

Before we explain our implementation, we will introduce some of the tools we use.

5.1.1 Ethereum smart contracts

We implement the 'on-chain' part of our construction using ethereum smart contracts. Ethereum is a commonly used public and programmable blockchain. It provides a set of computer instructions, that can be used to write smart contracts, as well as a a decentralized computing unit, the ethereum virtual machine (EVM), where the contracts are executed. Ethereum has a base currency, the ether (ETH), and uses a concept of accounts to associate a public key to an amount of ether controlled by the key. A specificity of ethereum is that smart contracts have their own accounts: users can transfer ether to contracts, and a contract can transfer ether from its accounts to users (or to other contracts). To avoid attackers using all of the resources of the EVM, ethereum uses a concept of gas. The gas is a fee that is included with every transaction. All resources (memory, computation) used for a transaction have a cost which is taken away from the gas and given to the ethereum nodes. If a transaction runs out of gas, the computations are reversed, but the gas used is not returned.

Solidity The ethereum community also produced high-level programming languages (solidity, vyper), that can be compiled to EVM programs. We used Solidity to define our contracts. Solidity has a syntax close to object-oriented programing languages, contracts have a member variables (the state), and methods (computation), that can be executed by specific transactions (we say that someone calls the method). Solidity contracts can emit `events`, which notifies the users that something happened. Deployed contracts are identified by an address (a unique 256 bit string), and can call eachother's method using this address. Solidity contracts are defined in source files with extension `.sol`.

5.1.2 zk-SNARKS

As a proof system, we use zk-SNARKS [6], which are non-interactive zero-knowledge proofs-of-knowledge. zk-SNARKS are secure in the common reference string (CRS) model, which means that they require a trusted setup. They have short proofs and fast verifications, which induces smaller transactions and reduced ethereum gas costs. More importantly, we can create zk-SNARKS to prove that a computation was correctly executed according to a fixed circuit, while not revealing all inputs (which are part of the witness). We can even produce proofs for tinyRAM (a random access machine with a limited instruction set) programs [7]. This allows us to produce zero-knowledge proofs for high-level computations, (e.g. Open and f in our construction).

5.1.3 Zokrates

One assumption we made about the contracts in Section 4.2, is that the contract is able to compute $b := \text{Verify}_\sigma(\langle z|c|y\rangle, \pi)$. Since Verify is not a native instruction of the EVM, we need to include its code as a part of smart the contract. We will use the Zokrates [8] toolbox for this purpose. Zokrates provides :

- A limited, high-level language to define computations with inputs that are specified as either `public` or `private`. Programs are written in files with extension `.zok`. The integer are limited to the type `field`, which represents values in $\mathbb{Z}_p$ for some prime p specific to the zk-SNARKs. We use `field` to represent 128-bit unsigned integers, as $p > 2^{128}$
- A compiler that reduces a Zokrates program to a zk-SNARK
- A setup algorithm that produces proving and verification keys
- A prover algorithm that takes in the proving key and the inputs, and produces an output, and a zero-knowledge proof. The statement to prove includes the `public` inputs and the output, and the witness includes the `private` inputs
- An export algorithm that takes in the verification key, and produces an ethereum smart contract, written in solidity, that can verify the proofs

5.1.4 Hash-Based Commitment

Our construction also needs a commitment scheme, with the particularity that the Open commitment is part of the computation verified by the zk-SNARKS, hence we need to write the check $\text{Open}(c, k) = \textit{secret}$ as a Zokrates program. Zokrates has a limited library, but it includes the SHA256 hash function, which can be used to build a hash-based commitment scheme. We then define our commitment scheme as in Figure **??**. The commitment c is a 256-bit string, and the key k is a 384-bit string. In zokrates we represent c with type `field[2]` , and k with type `field[3]`, by taking blocks of 128-bits for each `field`.

5.2 Library description

We have implemented our construction as a library [9], where all generic parts of the construction are implemented by the library and the developer only has to implement the parts that are specific to the target application. The library includes smart contracts, written in solidity, and javascript programs to help the owner and users of the contract.

5.2.1 Contract Architecture

We want our solution to abstract the way f is computed, such that the owner has little more to do than defining f, g, h, k and $l_{\text{fake}}, l_{\text{timeout}}$. We first break the code of the contract in three. A first contract is responsible for the computations of g, h, k and $l_{\text{fake}}, l_{\text{timeout}}$, we call it the `requester` contract. A second contract, is responsible for the computations of f, we call it the `holder` contract. The third contract, verifies the proofs sent by the owner, we call it the `verifier` contract. The three contracts are deployed together and know of each other's address.

Requester contract The requester has four methods: `start`, `callback`, `wrong_proof` and `timeout`.

- `start(data)` is called by the user which executes the pre-f computation (i.e. k and h) and obtains a 128-bit unsigned integer `z`, and calls `holder.requestComputation(z)`, obtains a value `id` and emits an event `Start(id)`
- `callback(id,y)` is called by the `holder` with the value $y := f(secret, z)$, executes the post-f computation (i.e. g), and emits an event `End(id, result)`, for some optional `result` generated by g
- `requester.wrong_proof(id)` is called by the `holder`, realizes l_{fake}, and emits an event `WrongProof(id)`
- `requester.timeout(id)` is called by the user, realizes l_{timeout}, and emits an event `Timeout(id)`

Verifier contract The verifier has one method: `verifyTx`, that runs the Verify algorithm, and returns the boolean result.

Holder contract The holder contract has a member variable `c`, initialized with the commitment of the *secret* parameter chosen by the owner. The holder has two methods : `requestComputation` and `answerRequest`.

- `requestComputation(z)` is called by the `requester`, it generates a unique value `id`, stores `(id, z)` in a table, emits an event `NewRequest(id, z)` and returns `id` to the requester
- `answerRequest(id,y,pi)` is called by the owner, it retrieves the tuple `(id,z)` from its table, and calls `verifier.verifyTx(⟨z|c|y⟩, pi)` to check that `y` is correct (i.e. $\langle z|c|y \rangle \in L_f$). If it is, then the contract calls `requester.callback(id, y)`, otherwise it calls `requester.wrong_proof(id)`

The `holder` is the same for all f, we can make its source code `holder.sol` an internal part of our library.

5.2.2 Local computations

Our library also includes programs that can be used to generate, deploy and interact with the smart contracts.

Setup program We've written a small setup program that is executed by the trusted third party, and produces the proving and verification key, for any given Zokrates program.

Owner program We require the owner to define f as a Zokrates program `F.zok`, and define g, h, k and $l_{\text{fake}}, l_{\text{timeout}}$ as part of the `requester` contract.

We wrote a program for the owner with three components as in Algorithm 2. Here *AddCommitCheck* modifies a zokrates program `F.zok` that computes `f(private secret, public z)`, to a new `F&C.zok` program that has additional inputs `(public c, private k)` and checks that $Open(c, k) = secret$ before computing f. *Link* modifies the 3 deployed contracts, such that that they know eachother's addresses.

Algorithm 2 Owner program

```
procedure GENERATE(F)
    F&C ← AddCommitCheck(F)
    p.key, v.key ← Trusted.Setup(F)
    v.sol ← Zokrates.export(v.key)
    return F&C, p.key, v.key, v.sol
end procedure
procedure DEPLOY(r.sol, v.sol, secret)
    r_addr ← Deploy(r.sol)
    v_addr ← Deploy(v.sol)
    (c,k) ← Commit(secret)
    d_addr ← Deploy(d.sol, c)
    Link(r_addr, v_addr, d_addr)
    return (c, k, r_addr, v_addr, d_addr)
end procedure
procedure LISTEN(c, k, s, h_addr, F&C, p.key)
    listener ← h_addr.listen(NewRequest)
    while true do
        id, z ← listener.waitEvent()
        y, pi ← Zokrates.Prove(F&C, p.key, s, z, c, k)
        h_addr.answerRequest(id,y,pi)
    end while
end procedure
```

User program We also defined a user program to interact with the `requester` contract, this allows the user to make calls to the contract as if it was a regular contract.

Algorithm 3 User program

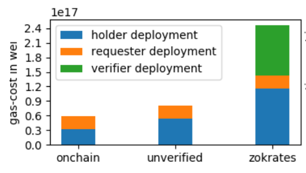
```
procedure MAKECALL(r_addr, input, nb_block)
    id ← r_addr.start(data)
    listener ← h_addr.listen(End(id,_))
    timeout, result ← listener.wait(nb_block)
    if timeout = true then
        r_addr.timeout(id)
        return TimeoutError
    else
        return result
    end if
end procedure
```

5.3 Details

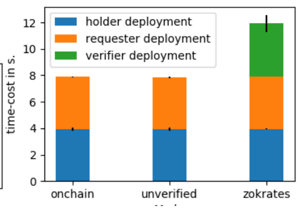
Here we have implemented a `requester` contract, that can only access f in an asynchronous manner, through a call back. This breaks the atomic nature of smart contract changes. The developer must put care in writing the contract, to make sure that the `Timeout`, `WrongProof` methods revert what needs to be reverted in case of failure. The developer also needs to make sure `start` cannot be called a second time before the `callback` of the first call was executed, to avoid interleaving of the calls. Another difference with regular ethereum smart contracts, is that now the owner needs to pay some gas fee to answer a request from the user. A solution is to write the `requester` contract in a way that any call to `start` needs to include a small reward in ether, which will be transfered to the owner when `callback` is called, to cover the gas fee. The user also needs to check that the owner correctly deployed the contracts, and that the `verifier` contract was correclty generated from the verification key outputted by the trusted third party.

6 Performance

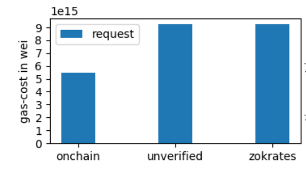
To analyze the added cost of our solution, we have compared 3 constructions. In the first one, called 'onchain' in the plots, all computations are done onchain, which corresponds to the construction of Section 4.1.1. In the second one, called 'unverified' in the plots, the computation of f is done offchain, but we don't use any verifications on the output, which corresponds to the construction of Section 4.1.2. The last one is our proposed construction, called 'zokrates' in the plots, described in Section 4.2 and implemented as described in Section 5. We wrote a dummy `requester` contract, and a dummy $f : (secret, x) \rightarrow secret + x$, which just takes an integer x as user input in (start) and outputs the value $secret + x$ as the result when `callback` is executed. We do the following experiment: first we use the owner program to generate and deploy the contracts and start the listener. Then we execute the user program to make a call to `start` and to wait for the result. For each action, we measured the time spent (in seconds), and the gas cost (in wei, or 10^{-18} ether) for each party (owner and user), we plotted the results in Figure 1. We set up a private test network with a block period of 2 seconds, and a gas price of 10^{14} wei.



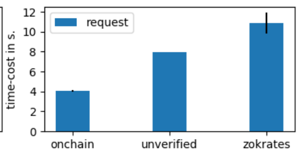
(a) Deployment gas



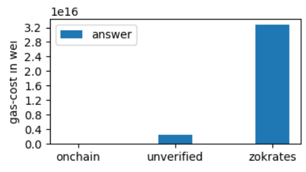
(b) Deployment time



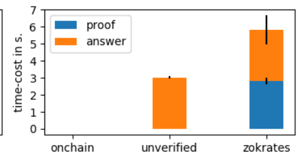
(c) Request gas



(d) Request time



(e) Answer gas



(f) Answer time

Figure 1: Performance comparison

Costs in Figures 1a, 1b, 1e and 1f are costs paid by the owner, while costs in Figures 1c and 1d are costs paid by the user. We can see that our construction increases significantly the costs for the owner. Though, as discussed in Section 5.3, the costs in Figure 1e can be covered by the user, through a reward system.

7 Future improvements

The main drawback of our construction is that it assumes a trusted setup producing the proving and verification keys. And this trusted setup needs to happen for each new f defined by the developer, as the zk-SNARKs we use are circuit-specific. A recent paper [10] has proposed constructions for universal zk-SNARKS, where one pair of keys can be used to verify the computation of any f (with a bounded circuit size). This means that the trusted setup would have to be called only once and the produced keys could be reused. Another solution would be to use zk-STARKS [11], which do not require any trusted setup. We would also like to explore other approaches, such as using secure multi-party computation protocols, obfuscation mechanisms, or secure hardware systems.

8 Conclusion

In this paper, we have proposed a construction that reconciles the public and incorruptible nature of a smart contract, with the sensitivity of some application data. We described a model of an ideal contract where some part of the contract's code is parameterized by a secret value, without leaking the secret value. We provide a construction that emulates this ideal contract, using a proof system and a commitment scheme. Our construction assumes that we can access a trusted setup for the proof system, and is secure against rational adversaries. We provided an implementation of this construction, using ethereum smart contracts and zk-SNARKS, which can be used as a library for a large set of applications.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger,"

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[3] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," 2015.

[4] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," 2014.

[5] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," 2018.

[6] J. Groth, "On the size of pairing-based non-interactive arguments," 2016.

[7] E. Ben-Sasson, A. Chiesa, D. Genkin, E. C. Tromer, and M. Virza, “Snarks for c: Verifying program executions succinctly and in zero knowledge,” 2013.

[8] J. Eberhardt and S. Tai, “Zokrates - scalable privacy-preserving off-chain computations,” July 2018.

[9] M. Thiercelin, “Offchainer library,” 2019. `https://github.com/marinthiercelin/offChainer`.

[10] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, “Marlin: Preprocessing zksnarks with universal and updatable srs,” 2019.

[11] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Scalable, transparent, and post-quantum secure computational integrity,” 2018.