Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services

Matteo Campanelli[1], Rosario Gennaro[1], Steven Goldfeder[2], and Luca Nizzardo[3,4]

[1] City College of New York, USA
{rosario@cs.ccny, mcampanelli@gc }.cuny.edu
[2] Princeton University, USA
stevenag@cs.princeton.edu
[3] IMDEA Software Institute, Madrid, Spain.
luca.nizzardo@imdea.org
[4] Universidad Politécnica de Madrid, Spain.

Abstract. Zero Knowledge Contingent Payment (ZKCP) protocols allow fair exchange of sold goods and payments over the Bitcoin network. In this paper we point out two main shortcomings of current proposals for ZKCP, and propose ways to address them.

First we show an attack that allows a buyer to learn partial information about the digital good being sold, without paying for it. This break in the zero-knowledge condition of ZKCP is due to the fact that in the protocols we attack, the buyer is allowed to choose common parameters that normally should be selected by a trusted third party. We implemented and tested this attack: we present code that learns, without paying, the value of a Sudoku cell in the "Pay-to-Sudoku" ZKCP implementation [18]. We also present ways to fix this attack that do not require a trusted third party.

Second, we show that ZKCP are not suited for the purchase of digital *services* rather than goods. Current constructions of ZKCP do not allow a seller to receive payments after proving that a certain service has been rendered, but only for the sale of a specific digital good. We define the notion of *Zero-Knowledge Contingent Service Payment* (ZKCSP) protocols and construct two new protocols, for either public or private verification. We implemented our ZKCSP protocols for *Proofs of Retrievability*, where a client pays the server for providing a proof that the client's data is correctly stored by the server.We also implement a secure ZKCP protocol for "Pay-to-Sudoku" via our ZKCSP protocol, which does not require a trusted third party.

A side product of our implementation effort is a new optimized circuit for `SHA256` with less than *a quarter* than the number of AND gates of the best previously publicly available one. Our new `SHA256` circuit may be of independent use for circuit-based MPC and FHE protocols that require `SHA256` circuits.

1 Introduction

The problem of fair exchange in which two parties want to swap digital goods such that neither can cheat the other has been studied for decades, and indeed it has been shown that fairness is unachievable without the aid of a trusted third party [23]. However, using Bitcoin or other blockchain-based cryptocurrencies, it has been demonstrated that fair-exchange can be achieved in a completely trustless manner. The previous results were not incorrect; a third party is definitely necessary, but the key innovation that Bitcoin brings to fair exchange is that the blockchain can fill the role of the trusted party, and essentially eliminate trust.

Consider Alice, an avid fan of brainteasers that has a Sudoku puzzle that she is stumped on. After trying for days to solve the puzzle, Alice gives up and posts the puzzle on an online message board proclaiming, "I will pay whoever provides me the solution to this puzzle". Bob sees this message, solves the puzzle, and wants to sell Alice the solution. But there's a problem: Alice wants

Bob to first provide the solution so that she can verify it's correct before she pays him, whereas Bob insists that he will not send Alice the solution until he has been paid. This is the classical problem of fair exchange: neither party wants to impart with its good before being sure that it will receive the other good in exchange.

To solve this problem, Alice and Bob could use a blockchain. Bitcoin and other blockchain-based cryptocurrencies allow one to post transactions that pay others and specify the conditions that need to be met in order for the money to be claimed. Alice can post a payment transaction to the blockchain that encodes the sudoku puzzle as well as the rules, and specifies that whoever provides the correct solution can claim the funds. In essence, the blockchain here is serving the traditional role of a trusted third party: Alice "deposits" funds in the blockchain, and the blockchain will only release those funds to Bob once he provides the correct solution.

While in theory this would work, there's one problem: Bitcoin's scripting language is limited, and does not allow one to directly specify arbitrary programs or conditions that are necessary to spend money. Zero Knowledge Contingent Payment (ZKCP) protocols [18,13,37] to allow fair exchange over the Bitcoin blockchain. The protocol makes use of a feature of the Bitcoin scripting language that allows one to create a payment transaction that specifies a value y and allows anyone who can provide a preimage k such that $\texttt{SHA256}(k) = y$ to claim the bitcoins [5].

In the ZKCP protocol, Bob knows a solution s and encrypts the solution to the puzzle using a key k such that $\texttt{Enc}_k(s) = c$. Bob also computes y such that $\texttt{SHA256}(k) = y$. He then sends Alice c and y together with a zero-knowledge proof that c is an encryption of s under the key k and that $\texttt{SHA256}(k) = y$. Once Alice has verified the proof, she creates a transaction to the blockchain that pays Bob n bitcoins, and specifies that Bob can only claim the funds if he provides a value k' such that $\texttt{SHA256}(k') = y$. Bob then published k and claims the funds. Alice, having learned k can now decrypt c, and hence she learns s.

When ZKCP was first introduced in 2011 it was only theoretical as there was no known efficient general purpose zero-knowledge protocol that could be used for the necessary proofs. Since then, however, advances have been made in this area, and there are now general-purpose *Succinct Non-Interactive Arguments of Knowledge (ZK-SNARK)* protocols that allow for the practical implementation of the necessary proofs. The protocol was refined to use SNARKs, and a sample-implementation for the Sudoku problem was also made available [18].

1.1 Breaking ZKCP

All NIZK proofs require a trusted party to generate the common reference string (CRS) for the production and the verification of the proof. The introduction of a third party, however, even to generate the parameters, is undesirable – recall that the entire point of ZKCP is to solve the fair exchange protocol in a completely trustless manner!

To eliminate the need for a trusted third party, proofs in ZKCP are made to convince one person – the buyer. It was natural therefore, for the buyer to serve as the trusted third party. Since the buyer trusts herself, she will be convinced of the correctness of the proofs. Using this observation, the ZKCP protocol specifies that the buyer should generate the CRS, and indeed the Sudoku implementation follows these guidelines.

But in ZKCP, there are two potential adversaries: the seller and the buyer. A malicious seller would try to cheat by producing a false proof that convinces the buyer to send her money even

[5] We are simplifying the protocol here. See Section 2.5 for full details.

though she will not receive the solution. Indeed, the current protocol protects against this attack. Since the buyer generates the CRS, the seller (prover) is unable to produce an incorrect proof that will be accepted by the buyer (verifier).

But the buyer can also be malicious. Indeed, if the buyer is able to break the zero-knowledge property of the proof, she may learn part of the solution from the seller even without paying! Intuitively, the buyer can modify the CRS such that the proof that the seller provides actually leaks some bits of the solution.

In the original SNARKs paper, it was assumed that the CRS was generated honestly, and indeed the proof of the zero-knowledge property made use of this fact [29]. When this assumption is violated, a malicious party can craft a CRS that allows it to break the zero-knowledge property and learn information about the witness. We note that *if the Prover checks that the CRS is "well formed"*, the SNARK in [29] remains *Witness Indistinguishable (WI)* – however this is not sufficient since in this case the witness is the Sudoku solution which is unique (and therefore even if the protocol is WI, information can be learned about the witness). We also note that with some additional more expensive checks, the SNARK in [29] remains ZK [7,28,1]. In ZKCP however neither of these checks are performed, and therefore, a malicious buyer can generate a malicious CRS that allows it to learn information from the seller's proof without paying. We show an attack on the "pay to Sudoku" protocol that proceeds along these lines, and we also provide code [21,22] that implements the attack and shows how one can break the zero-knowledge property and learn information in the sample sudoku code [18].

1.2 Fixing ZKCP

While issues arise when the verifier generates the CRS, the ZKCP high-level idea remains elegant and appealing. Therefore in Section 3.3 we discuss several ways to construct ZKCPs which do not require the help of a trusted party.

One way is to require that the CRS is constructed via a two-party secure computation protocol jointly by buyer and seller, a solution which allows them to "recycle" the CRS over several ZKCP executions. A similar approach was adopted by the designers of Zcash [42].

Another way is to use the notion of *Subversion-NIZK* [7], where ZK is preserved even when the verifier chooses the CRS. As we pointed out above, this requires the Prover to perform some "well formedness" checks on the CRS, which however can be somewhat expensive (as opposed to the minimal checks described in [29] to guarantee witness-indistinguishability).

At the end the simplest solution was to rely on a different type of protcol for *Zero-Knowledge Contingent Service Payments (ZKCSP)* which we describe below.

1.3 Zero-Knowledge Contingent Service Payments (ZKCSP): paying for digital services

We extend the idea of ZKCP to a new class of problems: paying for digital services.

Consider Alice, a user of a subscription online file storage service, FileBox. FileBox offers a service that for a small fee, it will provide a succinct proof-of-retrievability (PoR) [43] to its users demonstrating that all of that user's files are being stored. Alice would like to pay for this service, and thus we have a far exchange problem: Alice wants to pay once she receives proof that the files are being stored, whereas FileBox will only send the proof once it has been paid.

Notice that unlike the Sudoku example, Alice does not want any digital good (i.e. she doesn't want them to send her all the files). Instead, she just wants Filebox to demonstrate that they are indeed still storing the files.

The ZKCP protocol will fail in this case. If we try to apply this protocol and view the PoR as a "good" that Alice wants to receive, then the first step of a ZKCP protocol is to have FileBox create a proof that it has a PoR and send the encrypted PoR to Alice.

But a proof of a PoR is *itself* a PoR, and thus once Alice receives this zero-knowledge proof, she can abort the protocol as she already received the proof that she desired without paying.

As a second motivation, consider an online Bitcoin exchange that will provide proofs of solvency as a service for a fee. Often exchanges do not want to leak their inner details, and thus they may use Provisions [24], a privacy-preserving proof of solvency that shows that they are solvent without leaking their private accounting details. Bob stores his coins with this exchange and wishes to pay for the proof, and thus a fair exchange situation arrives.

Again, if we try to apply a ZKCP protocol, it will fail. If the exchange gives a zero knowledge proof of a proof of solvency, that itself is a proof of solvency, and Bob has received what he wants and does not need to pay.

To address this issue, we introduce **Zero-Knowledge Contingent Service Payments (ZKCSP)**. To illustrate, let's focus on the PoR example. Let v be the verification algorithm for the PoR. What Alice wants then is for FileBox to demonstrate that it knows m such that $v(m) = 1$.

Intuitively, our ZKCSP protocol works as follows: The prover outputs a string y and gives a zero-knowledge proof that attests to the following:

> If $v(m) = 1$, then I know the preimage of y under `SHA256`. But, if $v(m) = 0$, then the probability that I know a `SHA256` preimage of y is negligible.

We only provide the intuition here, but in Section 4 we show how we can efficiently construct proofs of this form. There we also prove that it is sufficient for the underlying SNARK to be *Witness-Indistinguishable*, and therefore the security of the protocol can be achieved even if the Verifier chooses the CRS, provided that the Prover performs the minimal checks required to guarantee witness indistinguishability.

Other Applications of ZKCSP. *Bug Bounty* is another interesting application for ZKCSP. A software company GoodCode Inc. releases a beta version of its new product and offers a reward for people who find bugs in the code. Normally a ZKCP would suffice: the seller proves in ZK that she found a bug, and the payment trigger the release of the code of the bug. But there may be situation where just the knowledge of the existence of a bug can be valuable to GoodCode (for example, knowing that there is a bug, they will delay release of the code, and avoid potential costly damages). In this case a ZKCSP must be used to make sure that GoodCode pays for such knowledge, and not just for the code of the bug.

In general any *auditing* or *compliance* application where the buyer is paying for this type of services will require a ZKCSP rather than a ZKCP.

ZKCP via ZKCSP. Since ZKCP is a special case of ZKCSP we can use our ZKCSP protocol to obtain a secure ZKCP scheme that does not require the prover to perform the expensive checks for "subversion-ZK" but only the minimal checks to guarantee WI.

1.4 Our Contributions

We make the following contributions:

Attacks and fixes on ZKCP: We show that the ZKCP protocol when instantiated as it is now, is insecure, and develop several concrete attacks that allow a malicious buyer to learn information about the witness without paying the seller. We implement our attack by writing code for a malicious buyer that interacts with the unmodified implementation of the seller [18], and learns information about the Sudoku solution. We discuss how to avoid these attacks and various possible solutions.

Zero-Knowledge Contingent Service Payments: We introduce this new notion, and provide protocols for ZKCSP in both the public and private verifier setting. Again using our PoR example, the public verifier setting is when one wants to perform the service for the general public. The private verifier setting is when one wants to provide the service only for a specific individual.

Implementation: We implemented and tested the ZKCP attack. We also implemented and tested our two new ZKCSP protocol, for the case of PoR, showing that they are feasible. Moreover we implemented a secure Pay-to-Sudoku ZKCP via our ZKCSP protocol. Our code is available here [21,22].

Improved `SHA256` circuit: In the process of our implementation of the ZKCSP protocols, we built a library for semi-automated boolean circuit generation. The `SHA256` circuit that we produce has 22,272 AND gates, whereas the best publicly available circuit had 90,825 AND gates [45]. We released our `SHA256` circuit together with our code as it may be of independent use for circuit-based MPC and FHE protocols that require `SHA256` circuits.

1.5 Other related work

In [6], Banasik et al. provide a ZKCP solution which avoids the use of NIZK by replacing the zero knowledge proof with an interactive protocol performed online. Moreover they avoid using hash-locked transactions since they claim that they are not standard and widely accepted in the Bitcoin network[6].

The protocol presented in [6] is vulnerable to the so-called *mauling* problem, where an adversary which knows the hash identifier T of a transaction is able to come up with a hash identifier T' that is semantically equivalent to T (i.e. spends the same transaction, has the same value, and the same inputs and outputs). As the authors of [6] point out, there are many Bitcoin software clients that cannot handle transactions appearing in the ledger with an hash identifier which is different from the original one (namely, the one with which they were posted) [4]. This effectively makes the transaction unredeemable, causing problems when creating Bitcoin contracts [3,4]. While the authors acknowledge the mauling problem, their scheme only addresses mauling due to malleability in ECDSA signatures, but does not address mauling due to changing the script.

An Ethereum-based contingent payment protocol is described by Tramer et al. in [46].

2 Preliminaries

2.1 Bitcoin and Ethereum

Bitcoin is a decentralized digital currency proposed in 2008 [38]. We present only the necessary background for this paper here, but refer the reader to [17] and [39] for a detailed treatment.

[6] To the best of our knowledge, this is not really a serious issue.

Bitcoins are typically associated with *addresses*, and an address is just a hash of a public key. To transfer bitcoins from one address to another, one crafts a *transaction* which lists one or more input addresses from which the funds will be taken and one or more output addresses to which the funds will be sent.

In order for a transaction to be valid, the transaction must be signed with the private keys corresponding to the input addresses, the sum of the outputs must be less than or equal to the sum of the inputs, and the inputs must not have previously been spent [38,12].

Signed transactions are broadcast to the Bitcoin peer-to-peer network. *Miners* check the validity of transactions and group them into *blocks*. Miners participate in a distributed consensus protocol that chains these blocks into an append-only global ledger called the *block chain*.

What we've described so far is a typical Bitcoin transaction, known as a *Pay-to-PubkeyHash* transaction. However, for each output, the transaction includes a *script* written in a stack-based programming language that specifies the conditions which must be met in order to spend this output in the future. For each input address, the transaction contains a reference to a previous transaction which listed this address as an output and specified the conditions required for it to be spent.

For a *Pay-to-PubkeyHash* transaction, the output script simply specifies an address and that in order to spend this output, one must sign with the associated private key. But Bitcoin scripts can be more complex as well. The Bitcoin scripting language has a limited set of op_codes or built-in functions that can be used to create scripts.

Using the The OP_ `SHA256` op_code, the Bitcoin scripting language supports *hash-locked* transactions that specify a value y and require that in order to spend this output, one must provide an x such that $\texttt{SHA256}(x) = y$.

A feature that was not initially included in the scripting language but introduced in 2012 is *Pay-to-ScriptHash* (P2SH) addresses. To redeem an output sent to a P2SH address, one must specify a script that hashes to this address, and then meet the conditions specified in the script[14].

Bitcoin scripts now also support `OP_CHECKLOCKTIMEVERIFY` and `OP_CHECKSEQUENCEVERIFY` op_codes. The op_codes allow one to specify execution paths in the spending scripts that can only be validated after some relative or absolute time. For example, one can send money to Alice's address and specify that after 24 hours if Alice has not redeemed the output, then Bob can claim it by signing with his private key[14].

Although miners will accept the validity of all transactions that Bitcoin supports when included in blocks that others mine, most miners will only include a smaller subset of those transactions in the blocks that they construct. These are referred to "standard" transactions, and historically, this mean that it was quite difficult to get nonstandard transactions onto the blockchain. In Bitcoin today, however, this is no longer an issue since almost all scripts are now considered standard when they are part of a P2SH transaction [2]. While Bitcoin's scripting language contains another useful op_code, it is not a Turing-complete language and is limited in practice. Ethereum is another cryptocurrency with a much more expressive scripting language that allows one to express arbitrary programs as conditions for spending money. As transactions can specify arbitrary scripts, there is no guarantee that they will ever halt. Each Ethereum transaction therefore contains *gas*, or money that is sent to the miner to run the transaction. Every computational step has a fixed gas cost, and the miner will only run the computation until it runs out of gas.

There is a global *gas limit* that specifies a maximum amount of gas that can be spent in a single block, and consequently in a single transaction. Although in theory Ethereum scripts can

support arbitrary programs, the current gas limits are quite restrictive and do not allow for complex computations.

2.2 Cryptographic Definitions

In the rest of the paper we will use the term *efficient algorithm* to denote probabilistic algorithms with a polynomial running time. Also we denote with $\mathsf{neg}(n)$ a *negligible* function defined over the integers, meaning that for every polynomial $P(\cdot)$ we have that there exists an integer n_P such that for all $n > n_P$, $\mathsf{neg}(n) \leq \frac{1}{P(n)}$.

Claw free function pairs We start by recalling the definition of *claw free function pairs*. Informally these are pairs of efficiently computable functions H_1, H_2 such that it is hard to find x_1, x_2 with $H_1(x_1) = H_2(x_2)$.

Definition 1. *Let $CFG(\cdot)$ be an efficient algorithm that on input of a security parameter 1^n outputs two functions $H_{1,n}$ and $H_{2,n}$ with domain and image $\{0,1\}^n$. We say that $CFG(\cdot)$ is a claw free function generator, and $H_{1,n}, H_{2,n}$ are a claw-free pair if*

- *$H_{1,n}$ and $H_{2,n}$ can be efficiently computed*
- *for any efficient algorithm $\mathcal{A}$ we have that for $(H_{1,n}, H_{2,n}) \leftarrow CFG(1^n)$*

$$\Pr[\mathcal{A}(H_{1,n}, H_{2,n}) = (x_1, x_2) \text{ } s.t. \\ H_{1,n}(x_1) = H_{2,n}(x_2)] \leq \mathsf{neg}(n)$$

Computational Indistinguishability Recall that two distributions are said to be computationally indistinguishable if no efficient algorithm can distinguish if elements are sampled according to one or the other distribution.

Definition 2. *Let $\mathcal{D}_{1,n}, \mathcal{D}_{2,n}$ be two (family of) distributions defined over $\{0,1\}^n$. We say that $\mathcal{D}_{1,n}, \mathcal{D}_{2,n}$ are computationally indistinguishable if for any efficient algorithm $\mathcal{A}$ we have that*

$$|\Pr[x \leftarrow \mathcal{D}_{1,n} \; ; \; \mathcal{A}(x) = 1] - \Pr[x \leftarrow \mathcal{D}_{2,n} \; ; \; \mathcal{A}(x) = 1]| \\ \leq \mathsf{neg}(n)$$

2.3 Fair Exchange

In this section we recall the definition of fair exchange following previous work in [5,35]. We have two parties Alice and Bob who want to exchange generic digital items. We know, due to a classic result of Cleve [23], that in the presence of malicious parties a fair exchange is impossible: one party will always have an advantage over the other. The traditional way to solve this problem is to rely on an *Arbiter*, a trusted third party (TTP), which is assumed to be honest and will help Alice and Bob exchange the items fairly. An *optimistic* fair exchange protocol involves the Arbiter only if one of the two parties does not behave honestly and complications arise. Two honest parties can exchange goods without involving the Arbiter.

The full definition in [5,35] involves also two other parties (assumed to also be honest): a *Tracker* and a *Bank*. The former is used to make sure that the goods exchanged by the parties are the correct ones, while the latter takes care of eventual payments and money exchanges. The verification of

the digital goods is executed by the Tracker in a trusted off-line phase where parties are provided with "verification keys" for the digital goods. For brevity's sake we are not going to describe this part and refer the reader to [5,35]. Instead we just assume that Alice's and Bob's inputs include these verification keys, together with some public parameters.

Definition 3. *A fair exchange protocol is a three-party communication protocol: Alice running algorithm A, Bob running an algorithm B, and the Arbiter running a trusted algorithm T. All parties run on input some public parameters PP, Alice runs on input f_A, V_A, Bob runs on input f_B, V_B, and the Arbiter runs on a input sk_T.*

We denote with $[a, b] \leftarrow [A(f_A, V_A), B(f_B, V_B), T(sk_T)]$ the event that at the end of the execution of the protocol Alice outputs a and Bob outputs b, where a, b can be $\perp$ meaning that the parties reject the execution (e.g. their output is not valid according to their verification key – we assume that the files $f_A, f_B \neq \perp$).

Completeness: A fair exchange protocol is complete if the execution of the protocol by honest parties results in Alice getting Bob's files and viceversa:

$$\mathsf{Pr}[\ [f_B, f_A] \leftarrow [A(f_A, V_A), B(f_B, V_B), T(sk_T)]] = 1$$

We say that a fair exchange is *optimistic* if the algorithm T is not invoked by the correct algorithms A and B.

Fairness: Intuitively, fairness states that, at the end of the protocol, either Alice and Bob get valid content (that is, content which passes the verification algorithm they were given by the Tracker), or neither Alice nor Bob get anything which passes the verification procedure. The above informal notion of fairness however does not capture the notion of partial information. It could be that a possibly malicious $\hat{B}$ learns something about a valid f_A while A outputs $\perp$. We strengthen the definition of fairness to capture the fact that if an honest party outputs $\perp$ then the other party learns no information. This is captured by a standard *simulation* definition. We say that a protocol is *fair* if for all efficient algorithms $\hat{B}$ there exists an efficient simulator $Sim_{\hat{B}}$ with oracle access to T such that the two distributions

$$[\perp, Sim^{T}_{\hat{B}}(f_B, V_B, V_A)]$$

and

$$[\perp, b] \leftarrow [A(f_A, V_A), \hat{B}(f_B, V_B), T(sk_T)]$$

are computationally indistinguishable. A dual condition must hold for any possibly malicious efficient $\hat{A}$.

2.4 Smart Contracts: Fair Exchange over Blockchains

Assume that the exchange is a typical marketplace transaction, where A is a seller, f_A is a digital good, B is a buyer, and f_B is money. If the money is implemented via a blockchain-based digital currency such as Bitcoin, then one can leverage the assumption that the blockchain is a trusted "entity" and use it as the arbiter in a fair exchange protocol. Since the blockchain is involved in the transaction anyway, to transfer the money from the buyer to the seller, we can dispense with the optimistic feature, and just use a protocol which always uses the arbiter.

These types of fair exchange over a blockchain have been called *smart contracts* and can be abstracted to work in the following way. The buyer B posts a transaction on the blockchain that basically says

> Transfer f_B coins to the party who presents a string f that satisfies the verification algorithm V_B

Then A can post a transaction that says

> Here is f_A that satisfies V_B. Transfer those f_B coins to my address.

This type of transactions can be implemented over blockchains with sufficiently rich *scripting languages:* recall that a *script* is the program that needs to be executed in order to spend an output on the blockchain. The scripting language in Ethereum [20,50] is sufficiently rich, and one can in theory run any program as part of a transaction, which allows the execution of arbitrary contracts. In practice, the gas cost and global gas limit the complexity of Ethereum scripts.

In the simplified transactions above, everybody will learn the object f being purchased by B. But this problem can be avoided by changing the verification procedure accordingly. B could request that the object f being purchased be encrypted under his public key, and published together with a non-interactive zero-knowledge proof that f satisfies the verification algorithm V_B. Note that the latter is an NP statement so (at least in theory) it can be proven in zero-knowledge. One interesting issue (which we discuss in Section 3) is how to actually implement this NIZK proof, and in particular the selection of the common reference string that is needed by such proofs.

This type of smart contracts that allow parties to buy and sell knowledge in a trustless manner have been named *Zero-Knowledge Contingent Payments (ZKCP)*, and as we will show below ZKCP protocols have been proposed over blockchain systems with more limited scripting language like Bitcoin [13].

2.5 Zero-Knowledge Contingent Payments: Fair Exchange over Bitcoin

The problem with the smart contract described above is that it is not possible to implement it directly in a Bitcoin transaction since the scripting language does not allow arbitrary verification procedures. Recall from Section 2.1 that a hash-locked transaction allows a party to redeem a transaction output if he/she produces the preimage (under `SHA256`) of a specific hashed value included in the original transaction.

Using hash-locked transactions the following construction was originally presented by Maxwell in 2011, and is now well known in the Bitcoin community [13,37]: Alice (seller) and Bob (buyer) engage in an offline phase, where Alice encrypts the string f_A with a key k (using any symmetric encryption scheme E, i.e. AES) and publishes $\hat{f} = E_k(f_A)$ and $s = \texttt{SHA256}(k)$ together with a ZK proof that $E^{-1}_{\texttt{SHA256}^{-1}(s)}(\hat{f})$ satisfies the verification procedure V_B. Again this is an NP statement and therefore can be proven in ZK. Since this interaction between Alice and Bob will not be posted on the blockchain, the proof could be performed interactively or non-interactively.

If the proof is correct, Bob then broadcasts the following transaction to be included in the blockchain:

> Transfer f_B Bitcoins to the party who presents a `SHA256` preimage of s and signs the transaction with pk_{Alice}. If this output is still unspent after n blocks, then the bitcoins can be claimed by pk_{Bob}.

At this point Alice can claim the coins by signing the transaction that publishes k, which in turn will allow Bob to recover the digital good f_A.

Note that the transaction that Bob posts requires that the seller provides both the preimage k as well as a signature. The reason that we also require a signature is to prevent a *front-running* attack in which Alice broadcasts k to the network to claim the funds, but before Alice's transaction is included in a block, some other party (perhaps the miner) sees k and uses it to claim the funds for themselves. To prevent this attack, the transaction requires Alice's signature as well, which nobody else can produce.

Also notice the second condition in the transaction that specifies that after a certain amount of time elapses, Bob can himself claim the output of this transaction. This is a *refund* clause that allows Bob to reclaim his output in case Alice decides not to post k. Without this clause, in the event that Alice decides not to complete the protocol and publish k, Bob's funds would be locked up and he would neither have his money nor the string f_A.

2.6 Example: Pay for Sudoku Solutions

When Maxwell first proposed ZKCP in 2011 it was only theoretical as there was no known efficient general purpose zero-knowledge protocol that could be used. But advances since then in zero knowledge protocols [29,25,10] have made this protocol feasible and indeed there is currently a publicly available implementation of ZKCP for purchasing Sudoku solutions. Using the template above, the string f_A is the solution of an $n \times n$ input Sudoku table (which also specifies the verification algorithm V_B). The main challenge of course is the implementation of the ZK proof that the decryption of $\hat{f}$ under the preimage of s is indeed a valid Sudoku solution for the input Sudoku table. They implemented this non-interactively, using the ZK *Succint Non-Interactive Arguments of Knowledge (ZK-SNARK)* based on Quadratic Arithmetic Programs [29,40], using the libsnark library [8,11].

As with all NIZK proofs, QSP-based ZK-SNARGs require a common reference string (CRS) for the production and the verification of the proof[7]. Such CRS should be selected by a trusted party in advance, which is obviously non-ideal for ZKCP. The entire premise of ZKCP is to perform fair-exchange over the blockchain in a trustless manner, and introducing a trusted third party would largely defeat the purpose.

To get around this, it was noticed that unlike proofs which are produced to be verified by the public, the ZK-proof in ZKCP only need to convince a single person – the buyer. In ZKCP it was therefore proposed that the buyer (i.e. the verifier) generate the CRS, to ensure that the seller could not cheat.

However, having the buyer generate the CRS is problematic as it only protects against a *soundness adversary* but not a *zero-knowledge adversary*. With regards to the proof's *soundness* property, the seller is the adversary as the seller would benefit from producing an incorrect proof. However, with regards to the proof's *zero-knowledge* property, the buyer is the adversary as the buyer would benefit from learning some information about f_A without paying for it. If one generates the CRS maliciously, and (as we show below) the CRS is not checked for "correctness", they can break both soundness and zero-knowledge.

Because the ZKCP protocol does not check the correctness of the CRS it only ensures that the seller can not cheat, but it allows the buyer to cheat and extract information about the witness f_A without paying for it. In the next section we use this fact to show a concrete attack on the ZKCP protocol that leaks information about the value of a Sudoku cell before the buyer pays for the solution.

[7] In the SNARKs literature, the CRS is sometimes referred to as the *proving key* and the *verifying key*.

3 Attacks on ZKCP with untrusted CRS

In this section we show how allowing the Verifier to choose the CRS in the QAP-based SNARK leads to a loss of the Zero-Knowledge property. While it is a well known fact in the cryptographic literature that a trusted CRS is needed for zero-knowledge, the point of this section is to demonstrate this insecurity by developing concrete attacks that allow one to learn information in the "Pay-to-Sudoku" implementation, where the Verifier does indeed set the CRS. Through our attack, the Verifier is able to verify if a particular guess for a Sudoku cell is correct or not. This obviously break the fairness of the protocol (as defined in Section 2.3) since the buyer learns partial information about the seller's input.

First we recall how Quadratic Arithmetic Span programs work, since they are the proof backbone of the libsnark library used in the implementation. Then we show our attacks, and describe our implementation of the malicious verifier.

3.1 ZK-SNARKs from Quadratic Arithmetic Programs

We recall here the notion of Quadratic Arithmetic Programs (QAPs) [29,40], using the notation of Ben-Sasson et al. [11].

Definition 4 ([29]). *A QAP $\mathcal{Q}$ over a field $\mathbb{F}$ is defined by three sets of polynomials $A := \{A_i(x)\}_{i=0}^m, B := \{B_i(x)\}_{i=0}^m, C := \{C_i(x)\}_{i=0}^m$ and a target polynomial $Z(x)$. If we take a function $f : \mathbb{F}^n \to \mathbb{F}^{n'}$, then we say that $\mathcal{Q}$ computes f if, given a valid assignment $(c_1, \ldots, c_{n+n'})$ of inputs and outputs of f, there exist coefficients $(c_{n+n'+1}, \ldots, c_m)$ such that $Z(x)$ divides the following polynomial*

$$p(x) := \big(A_0(x) + \sum_{k=1}^{m} c_k \cdot A_k(x)\big) \cdot \big(B_0(x) + \sum_{k=1}^{m} c_k \cdot B_k(x)\big) + \\ -\big(C_0(x) + \sum_{k=1}^{m} c_k \cdot C_k(x)\big)$$

In other words there must exists a polynomial $H(x)$ such that $p(x) = H(x) \cdot Z(x)$. We refer to m and the degree of $Z(x)$ as the size and the degree of $\mathcal{Q}$ respectively.

To build a QAP for a function f, we use an arithmetic circuit $\mathcal{C}$ representing f; we then pick a distinct root r_g for any of its multiplicative gates. Then, we build the target polynomial as $Z(z) := \prod_g (z - r_g)$, and we label each input of the circuit and each output of a multiplicative gate with an index $i \in [m]$ (grouping together all the additive gates). We define the polynomials A, B, C in a way that they respectively encode the left, right and output wire of each gate: for example, $B_i(r_g) = 1$ if the i-th wire of the circuit is a right input wire of the gate g, and $B_i(r_g) = 0$ (and similarly with A and C with left input and output wires respectively). So, for any gate g and its root r_g, the condition above can be seen as:

$$\big(\sum_{k=1}^{m} c_k \cdot A_k(r_g)\big) \cdot \big(\sum_{k=1}^{m} c_k \cdot B_k(r_g)\big) = \\ = \big(\sum_{k \in I_L} c_k \cdot A_k(r_g)\big) \cdot \big(\sum_{k \in I_R} c_k \cdot B_k(r_g)\big) = c_g C_k(r_g) = c_g$$

which basically says that the output of a multiplication gate is the multiplication between the values on the left and the right inputs wire of the gate itself. Following the notation of [11], it is now possible to use QAPs to build zk-SNARKs, as in [29,40]:

Public Parameters: $\mathsf{pp} := (r, e, \mathcal{P}_1, \mathcal{P}_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ where $\mathbb{G}_1 := \langle \mathcal{P}_1 \rangle, \mathbb{G}_2 := \langle \mathcal{P}_2 \rangle, \mathbb{G}_T$ are groups of prime order r and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a pairing.

Key Generation The key generation procedure is composed by several steps, it takes in input a circuit $\mathcal{C} : \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^\ell$ and outputs a proving key pk and a verification key vk.

1. Compute (A, B, C, Z) with respect to the circuit $\mathcal{C}$ and extend $A := \{A_i(x)\}_{i=0}^m, B := \{B_i(x)\}_{i=0}^m, C := \{C_i(x)\}_{i=0}^m$ via $A_{m+1} = B_{m+2} = C_{m+3} = Z$, $A_{m+2} = A_{m+3} = B_{m+1} = B_{m+3} = C_{m+1} = C_{m+2} = 0$.
2. Sample $\tau, \varphi_A, \varphi_B, \alpha_A, \alpha_B, \alpha_C, \beta, \gamma \xleftarrow{\$} \mathbb{F}_r$
3. For $i = 0, \ldots, m+3$, let

   $pk_{A,i} := A_i(\tau)\varphi_A\mathcal{P}_1, \;\; pk'_{A,i} := A_i(\tau)\alpha_A\varphi_A\mathcal{P}_1$

   $pk_{B,i} := B_i(\tau)\varphi_B\mathcal{P}_2, \;\; pk'_{B,i} := B_i(\tau)\alpha_B\varphi_B\mathcal{P}_1$

   $pk_{C,i} := C_i(\tau)\varphi_C\mathcal{P}_1, \;\; pk'_{C,i} := C_i(\tau)\alpha_C\varphi_A\varphi_B\mathcal{P}_1$

   $pk_{K,i} := \beta(A_i(\tau)\varphi_A + B_i(\tau)\varphi_B + C_i(\tau)\alpha_C\varphi_A\varphi_B)\mathcal{P}_1$

   and for $i = 0, \ldots, d$ let $pk_{H,i} := \tau^i\mathcal{P}_1$. Set
   $pk := (\mathcal{C}, pk_A, pk'_A, pk_B, pk'_B, pk_C, pk'_C, pk_K, pk_H)$.
4. Let

   $vk_A := \alpha_A\mathcal{P}_2, \; vk_B := \alpha_B\mathcal{P}_1, \; vk_C := \alpha_C\mathcal{P}_2$

   $vk_\gamma := \gamma\mathcal{P}_2, \; vk^1_{\gamma\beta} := \gamma\beta\mathcal{P}_1, \; vk^2_{\gamma\beta} := \gamma\beta\mathcal{P}_2$

   $vk_Z := Z(\tau)\varphi_A\varphi_B\mathcal{P}_2$

   $\{vk_{IC,i}\}_{i=0}^n := \{A_i(\tau)\varphi_A\mathcal{P}_1\}_{i=0}^n$.

   Set
   $vk := (vk_A, vk_B, vk_C, vk_\gamma, vk^1_{\gamma\beta}, vk^2_{\gamma\beta}, vk_Z, vk_{IC})$.
5. Output (pk, vk)

Prover: On input a proving key pk, an input $x \in \mathbb{F}_r^n$, a witness $a \in \mathbb{F}_r^h$, it outputs a proof π which is computed as follows:

1. Compute (A, B, C, Z) with respect to the circuit $\mathcal{C}$.
2. Compute the QAP witness $s \in \mathbb{F}^m$ with respect to $\mathcal{C}, x, a$.
3. Sample $\delta_1, \delta_2, \delta_3 \xleftarrow{\$} \mathbb{F}_r$.
4. Compute the polynomial

$$H(z) := \frac{A(z)B(z) - C(z)}{Z(z)}$$

   where

$$A(z) := A_0(z) + \sum_{i=1}^{m} s_i A_i(z) + \delta_1 Z(z),$$
$$B(z) := B_0(z) + \sum_{i=1}^{m} s_i B_i(z) + \delta_2 Z(z),$$
$$C(z) := C_0(z) + \sum_{i=1}^{m} s_i C_i(z) + \delta_3 Z(z).$$

and represent $H(z)$ as $(h_0, \ldots, h_d) \in \mathbb{F}_r^{d+1}$

5. Set

$$\tilde{pk}_A := (0^n, pk_{A,n+1}, \ldots, pk_{A,m+3})$$
$$\tilde{pk}'_A := (0^n, pk'_{A,n+1}, \ldots, pk_{A,m+3}).$$

6. Let $c := (1, s, \delta_1, \delta_2, \delta_3) \in \mathbb{F}_r^{4+m}$, compute

$$\pi_A := \langle c, \tilde{pk}_A \rangle, \quad \pi'_A := \langle c, \tilde{pk}'_A \rangle,$$
$$\pi_B := \langle c, pk_B \rangle, \quad \pi'_B := \langle c, pk'_B \rangle,$$
$$\pi_C := \langle c, pk_C \rangle, \quad \pi'_C := \langle c, pk'_C \rangle,$$
$$\pi_K := \langle c, pk_K \rangle, \quad \pi_H := \langle h, pk_K \rangle.$$

7. Output $\pi := (\pi_A, \pi'_A, \pi_B, \pi'_B, \pi_C, \pi'_C, \pi_K, \pi_H)$.

Verifier: On input a verification key vk, an input $x \in \mathbb{F}_r^n$ and a proof π, the verifier proceeds as follows:

1. Compute $vk_x := vk_{IC,0} + \sum_{i=1}^{m} x_i vk_{IC,i} \in \mathbb{G}_1$.
2. Verify validity of knowledge commitments for A, B, C by checking:

$$e(\pi_A, vk_A) = e(\pi'_A, \mathcal{P}_2), \quad e(\mathsf{vk}_B, \pi_B) = e(\pi'_B, \mathcal{P}_2), \quad e(\pi_C, vk_C) = e(\pi'_C, \mathcal{P}_2).$$

3. Verify that the same coefficients were used by checking:

$$e(\pi_K, vk_\gamma) = e(vk_x + \pi_A + \pi_C, vk^2_{\gamma\beta}) \cdot e(vk^1_{\gamma\beta}, \pi_B).$$

4. Check QAP divisibility

$$e(vk_x + \pi_A, \pi_B) = e(\pi_H, vk_Z) \cdot e(\pi_C, \mathcal{P}_2).$$

5. Output 1 (accept) if and only if all the above checks are satisfied.

3.2 Learning Information by modifying the CRS

If a possibly malicious Verifier is allowed to set the CRS (as in the "Pay to Sudoku" (PtS) code [18]), then there are a variety of attacks that can be attempted to learn information about the Sudoku solution during the offline phase of the ZKCP (and therefore before the payment phase is completed).

Changing the Circuit This is the easiest attack to consider. Recall that the CRS of a QAP-based SNARK consists of an encoding of a QAP encoding of the function f that verifies the NP witness held by the prover. A malicious verifier could just replace the CRS with the QAP encoding of a modified function $\tilde{f}$ whose output directly leaks the needed information. In other words, the sets

of polynomials A, B, C and C, and the polynomial Z would be modified to $\tilde{A}, \tilde{B}, \tilde{C}, \tilde{Z}$. Nevertheless this trivial attack does not work in a libsnark implementation of QAP-based SNARKs. The reason is that the QAP-encoding of a function f is a deterministic process, and in libsnark both prover and verifier compute the polynomials A, B, C, Z on their own directly from a description of the function f, and this leads to a straightforward detection of any change.

Learning one wire is sufficient We now point out that in the PtS implementation, for every Sudoku cell, there are n wires $w_1, \ldots, w_n$ in the circuit $\mathcal{C}$ used in the SNARK, such that $w_j = 1$ if the cell is set to j in the solution, while all the other wires related to that cell are set to 0. Therefore learning the value of the wire w_j will allow us to learn if that particular cell is set to j or not. Recall from the previous section that the value of the wires of $\mathcal{C}$ are the coefficients c_i used to compute the linear combinations so it is sufficient to learn c_j. Note also that c_j can only assume a binary value. We now focus on attacks that allow us to compute a single coefficient c_j.

Choosing τ as one of the roots of Z In the correct CRS generation, τ is chosen at random in the field $\mathbb{F}_r$. It turns out that if one selects τ as one of the roots of $Z(x)$, then τ is also the root of all the polynomials A, B, C except for one of them, say $B_j(x)$, for which $B_j(\tau) \neq 0$. In this case the component π_B of the proof produced by the prover reveals the value $\gamma_j = c_j \phi_B \mathcal{P}_2$ which allows to recover c_j since it can only assume a binary value. This attack is not detected in libsnark on the prover side (it would be easily detected by checking the public key pk and see if it contains the identity in either $\mathbb{G}_1$ or $\mathbb{G}_2$, but this check is not performed in libsnark). However the attack does not work in the "Pay-to-Sudoku" ZKCP for a very interesting reason. The prover code actually produces the "wire value leaking" proof π_B without an error, but then before sending it out to the Verifier, the PtS code has the Prover run a verification of its own proof π. This verification fails because the polynomial $H(x)$ is computed by dividing via the polynomial $Z(x)$ and so when evaluated at τ the QAP divisibility check fail. Moreover, because of an optimization step of the verification procedure that does not expect to compute a pairing operation where the input in $\mathbb{G}_2$ is the identity, the proof fails even before getting to the QAP divisibility check (this will happen in the verification equation since $Z(\tau) = 0$ implies that $vk_Z = 0\mathcal{P}_2 = 0 \in \mathbb{G}_2$ and this value is placed in the $\mathbb{G}_2$ pairing input of one of the verification equations).

Setting all the pk equal to the identity, except for one wire This is the attack that works. The attack is described in detail below, but here we give an informal explanation. Here τ is selected at random, but it is not used to evaluate the polynomials. Similar to the attack above, the malicious verifier will set all the $pk_A, pk'_A, pk_C, pk'_C \in \mathbb{G}_1$ equal to 0 instead of setting them as the evaluation "in the exponent" of the polynomials A, C evaluated at τ. Similarly $pk_{B,i} = 0 \in \mathbb{G}_2$ and $pk'_{B,i} = 0 \in \mathbb{G}_1$ for all $i \neq j$ and $pk_{B,j} = \varphi_B \mathcal{P}_2$, $pk'_{B,j} = \alpha_B \varphi_B \mathcal{P}_1$ for known α_B, φ_B. By setting the pk, pk' values this way, the proof π will reveal the value γ_j as above, and therefore the value c_j. Since the prover checks its own proof before releasing it, we need to make sure that the proof verifies. We do that by setting $pk_{H,i} = 0 \in \mathbb{G}_1$ which will force the value π_H produced by the prover to be $\pi_H = 0 \in \mathbb{G}_1$. Moreover since all the identities are now only in the group $\mathbb{G}_1$, the error caused by the optimization in the libsnark implementation will not appear and indeed the proof is produced by the Prover (seller) and sent out to the Verifier (buyer), who will recover the value c_j.

More in details:

Public Parameters: Both the buyer and the prover get the public parameters and $\mathsf{pp} := (r, e, \mathcal{P}_1, \mathcal{P}_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \mathcal{C})$ which include the description of the circuit $\mathcal{C}$.

Key Generation: The buyer takes the circuit $\mathcal{C} : \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^\ell$ and outputs a proving key pk and a verification key vk as follows:

1. Honestly computes (A, B, C, Z) with respect to the circuit $\mathcal{C}$, where $A := \{A_i(x)\}_{i=0}^m, B := \{B_i(x)\}_{i=0}^m, C := \{C_i(x)\}_{i=0}^m$.
   Now he extends A, B, C via
   $A_{m+1} = A_{m+2} = A_{m+3} = 0$,
   $B_{m+1} = B_{m+2} = B_{m+3} = 0$,
   $C_{m+1} = C_{m+2} = C_{m+3} = 0$.
2. Sample $\tau, \varphi_A, \varphi_B, \alpha_A, \alpha_B, \alpha_C, \beta, \gamma \xleftarrow{\$} \mathbb{F}_r$.
3. For $i = 0, \ldots, m+3$, let
   $pk_{A,i} := 0 \in \mathbb{G}_1$, $pk'_{A,i} := 0 \in \mathbb{G}_1$,
   $pk_{B,i} := 0 \in \mathbb{G}_2$ for all $i \neq j$ and $pk_{B,j} := \varphi_B \mathcal{P}_2$,
   $pk'_{B,i} := 0 \in \mathbb{G}_1$ for all $i \neq j$ and
   $pk'_{B,j} = \alpha_B \varphi_B \mathcal{P}_1$,
   $pk_{C,i} := 0 \in \mathbb{G}_1$, $pk'_{C,i} := 0 \in \mathbb{G}_1$,
   $pk_{K,i} := 0 \in \mathbb{G}_1$ for all $i \neq j$, $pk_{K,j} := \beta \varphi_B \mathcal{P}_1$.
   For $i = 0, \ldots, d$ let $pk_{H,i} := 0 \in \mathbb{G}_1$, and set
   $pk := (pk_A, pk'_A, pk_B, pk'_B, pk_C, pk'_C, pk_K, pk_H)$.
4. Let $vk_A := \alpha_A \mathcal{P}_2$, $vk_B := \alpha_B \mathcal{P}_1$, $vk_C := \alpha_C \mathcal{P}_2$
   $vk_\gamma := \gamma \mathcal{P}_2$, $vk^1_{\gamma\beta} := \gamma\beta \mathcal{P}_1$, $vk^2_{\gamma\beta} := \gamma\beta \mathcal{P}_2$
   $vk_Z := Z(\tau)\varphi_A\varphi_B \mathcal{P}_2$, $\{vk_{IC,i}\}_{i=0}^n := \{0 \in \mathbb{G}_1\}_{i=0}^n$ and set
   $vk := (vk_A, vk_B, vk_C, vk_\gamma, vk^1_{\gamma\beta}, vk^2_{\gamma\beta}, vk_Z, vk_{IC})$.
5. Output (pk, vk)

It is not hard to see that all the verification equations are satisfied, and that the proof leaks the value c_j. If used against the PtS code for contingent payments for Sudoku solutions, this attack allows to find out the value for a Sudoku cell with probability 1/9. We provide an implementation for the attack above; see Section 5.1 for more details.

3.3 Countermeasures

In this section we show some possible countermeasures to our attack above.

Checking the CRS. As already discussed in the original paper on QSP/QAP [29] the prover can check that the CRS is "correctly formed" and in this case the protocol is *witness indistinguishable (WI)* [27]. In the QAP-based SNARK described in the previous section, it is sufficient that the prover/seller checks that

– The polynomials A, B, C, Z are well formed with respect to the circuit C.

- The elements $pk_{A_{m+1}}$, $pk'_{A_{m+1}}$, $pk'_{B_{m+2}}$, $pk_{C_{m+3}}$, $pk'_{C_{m+3}}$ are not equal to $0 \in \mathbb{G}_1$ and the element $pk_{B_{m+2}}$ is not equal to $0 \in \mathbb{G}_2$
- All the elements $pk_{H,i}$ are not $0 \in \mathbb{G}_1$.
- The element vk_Z is such that $vk_Z \neq 0 \in \mathbb{G}_2$.

since this will guarantee that the proof is a uniformly distributed random value no matter what witness is used (see [29]). This could be a good option for some applications of ZKCP, but unfortunately not for the PtS application since a Sudoku puzzle typically has only one solution and witness indistinguishability guarantees only that proofs "look the same" no matter what witness is used in the case that there are two or more such witnesses. It does not guarantee that no knowledge is leaked about a unique witness.

Subversion Resistant ZK. In a recent paper Bellare et al. introduce the notion of Subversion Zero Knowledge [7], i.e. the ability to prove ZK even when the CRS is maliciously selected by the verifier. Note that given some well known impossibility results [31,30], the notion of ZK obtained in this case is somewhat weak (ZK does not hold with respect to arbitrary auxiliary inputs the verifier might have). One could then run a ZKCP with a subversion resistant ZK protocol.

The proposed solution in [7] is not a SNARK (the proof is not succinct), but it is not hard to see that their techniques extend to the original QSP/QAP protocol in [29]. Indeed subversion-ZK can be obtained as long as the above "WI checks" are performed and the value τ can be extracted by the simulator from the Verifier when it produces the CRS. Following the approach in [7] one could use a "knowledge of exponent" type of assumption to extract τ after checking that each $pk_{H,i}$ is correct, i.e $pk_{H,i} = \tau^i \mathcal{P}_1$. In the original QSP/QAP protocol in [29], where $\mathbb{G}_1 = \mathbb{G}_2$, this can be checked using the bilinear map by checking that $e(\mathcal{P}_1, pk_{H,i}) = e(pk_{H,1}, pk_{H,i-1})$ for all i. The above intuition is actually formalized in [28] (a different subversion-ZK SNARK is presented in [1]).

Note that this check requires the computation of $\Theta(m)$ bilinear maps, a much more expensive task than the simple checks required for WI. Moreover it is not clear if those techniques extend to Pinocchio [40], the optimized version of the QSP/QAP protocol used by Libsnark [8], since in that case $\mathbb{G}_1 \neq \mathbb{G}_2$ and the above check cannot be performed. **Our experimental results suggests that running the subversion-resistant checks of [28] for the pay-to-sudoku example would take more than an hour on our benchmark machine** (see Section 5.2 for full details).

So to summarize, one could obtain (a weak non-aux input notion of) zero-knowledge by using subversion resistant ZK, but it would require major changes in the current implementation of ZKCP protocols, and increase the computation required of the Prover.

Distributed Generation of the CRS. Another possible solution is to have buyer and seller run a two-party secure computation protocol to compute the CRS together. Note that due to the algebraic structure of the CRS, this could be done via a much more efficient ad-hoc protocol, rather than say a generic solution such as Yao's protocol. A similar approach was followed by the designers of Zcash [19,9] to remove a trusted generation of the CRS in their the QAP-based SNARKs[8].

Using Contingent Payments for Services. At the end the best solution in our opinion is to use the protocol for ZK Contingent Service Payments that we describe in the next Section.

[8] We also point out that the CRS in their case is an "extended" version of the Pinocchio CRS, where both $\tau^i \mathcal{P}_1$ and $\tau^i \mathcal{P}_2$ appear in the CRS. This allows *anybody* to verify the correctness of the CRS via bilinear maps. Moreover, even if this CRS was computed by a single malicious party, rather than distributively, subversion ZK is guaranteed since the value τ can be extracted via a "knowledge of exponent" type of assumption.

4 Contingent Service Payments

In this section we discuss Contingent Payments for services such as auditing. Consider for example the case where Alice (the seller) is a data storage company, and Bob (the buyer) is a customer. Bob will store his files with Alice, and will pay her for this service. Assume that the contract between Bob and Alice is that periodically Alice will prove to Bob that his files are all correctly stored, and upon that proof Bob will pay the contracted rent.

There are several cryptographic protocols that allow a data storage provider to efficiently prove the integrity of the stored files to a customer. These are known under the name of *Proofs of Retrievability (PoR)* [32] and they all work by requiring that the prover shows the possession of a certain number of blocks previously authenticated by the client[9].

This can be achieved easily using a smart contract over a blockchain with a sufficiently rich scripting language. The client simply posts a transaction that pays whoever shows possession of such authenticated blocks. When the server posts those blocks, it will receive payment and the client will be assured all its files are still correctly stored.

But it does *not* work using the generic blueprint for ZKCP over Bitcoin described earlier. Indeed that blueprint requires the Server to prove possession of such blocks during the offline phase, but at this point the client has the desired knowledge (the server knows those blocks, therefore it must know the entire file) without having paid for it, and indeed the client does not have to post the payment transaction on the Bitcoin blockchain. The reason is that the ZKCP blueprint is designed for the sale of digital goods, but not digital services. In the ZKCP protocol described earlier the prover proves possession of a certain string, without revealing it, and the payment is contingent on the disclosure of the string. But in this case it's the proof of possession itself that is the valuable "service" desired by the buyer.

In the following we show how to design a ZKCP for digital services such as auditing and the other applications discussed in the Introduction.

4.1 Defining ZKCP for Services (ZKCSP)

We are looking for a protocol where a server A proves to a client B that he (the server) knows s such $f(s) = 1$ for an efficiently computable verification function $f : \{0,1\}^* \to \{0,1\}$ and needs to be paid for this information. Informally the properties that we would like to have are

- If a possibly malicious $\hat{A}$ is paid then $\hat{A}$ must "known" a value s such that $f(s) = 1$;
- If a possibly malicious $\hat{B}$ does not pay then $\hat{B}$ has learned no information
- additionally, a possibly malicious $\hat{B}$ who pays, learns only that A knows s such that $f(s) = 1$ and nothing else.

The latter condition can be relaxed in some settings, but by enforcing it we really limit the knowledge disclosure from A to B to a minimum.

We use a trusted party T which models a blockchain. T maintains a ledger of all the "coin balances" of each party. Moreover T accepts messages from A and B of only two types, and will execute the instructions honestly:

[9] Trivially the client can ask the prover to send back all the data originally stored and authenticated by the client, but this is not efficient. PoR protocols allow the server to prove that *all* the data is there by showing only a small number of blocks authenticated by the client (see appendix).

Contingent Payments from B which are of the form

> Transfer m of my coins to a party who publishes x such that when you run this program P on x you get $P(x) = 1$

In this case T checks that B has more than m coins and if so accepts the message and publishes it on the blockchain, otherwise it rejects it.

Redemption Payments from A which are of the form

> Transfer m coins to my account since I am publishing x such that when you run P on x you get $P(x) = 1$

In this case T checks that there is a previously accepted Contingent Payment message that refers to this program P, and that $P(x) = 1$. If so it will posts the message to the blockchain and will deduct m coins from the balance of the party B who posted the message, and adds has those m coins to the balance of A.

A Zero-Knowledge Contingent Service Payment (ZKCSP) protocol is a three party protocol defined by the interactive machines A, B, T where A runs on a private input s, and all parties run on public input a function f. We define the view of B, $View_{\hat{B}}(s, f)$ as his coin tosses together with all the messages exchanged during the protocol:

$$View_{\hat{B}}(s, f) := [Coins_{\hat{B}}||Messages[A(s, f), \hat{B}(f), T(f)]|| Out(A(s, f), \hat{B}(f), T(f)]$$

We say that (A, B, T) is a secure ZKCSP protocol if the following conditions are satisfied (all parties run on a security parameter 1^n)

Extraction For any possibly malicious efficient $\hat{A}$, if at the end of the protocol $\hat{A}$'s balance increases with non-negligible probability, then there exists an efficient extractor $Ext_{\hat{A}}$, which outputs a string $\hat{s}$ such that $f(\hat{s}) = 1$;

Zero-Knowledge For any possibly malicious efficient $\hat{B}$, there exists an efficient simulator $Sim_{\hat{B}}$ which on input f outputs a distribution which is computationally indistinguishable from $View_{\hat{B}}(s, f)$;

4.2 A ZKCSP Protocol

Given that s is basically the witness of an NP statement, it is possible to construct NIZK proofs of knowledge for it ([41] and the more recent literature on SNARKs [29,40,11]). If V is the program that verifies this NIZK proof (using a trusted CRS) then it is easy to implement a ZKCSP over any blockchain with sufficiently rich scripting languages such as Ethereum. The client B will post the transaction

> Transfer m of my coins to a party who publishes a proof π such that $V(\pi) = 1$

Once A publishes π she will get paid and B has confidence that A really knows s (with the simulation and extraction procedures being guaranteed by the simulation and extraction procedure of the NIZK used in the protocol). The question then is how to implement this over more limited scripting languages, including Bitcoin. What follows is a protocol where the program P associated with "payment transactions" can only be of the form "find a `SHA256` preimage of a specified value", i.e. hash-locked transactions.

Let H be a function $\mathsf{H}\{0,1\}^* \to \{0,1\}^{256}$ (i.e. like $\texttt{SHA256}$). Consider the following function

$$\mathcal{F}_{f,\mathsf{H}} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^{256}$$

defined as follows

$$\mathcal{F}_{f,\mathsf{H}}(s,r) = \begin{cases} \texttt{SHA256}(r) \text{ if } f(s) = 1 \\ \mathsf{H}(r) \text{ otherwise.} \end{cases} \tag{1}$$

We are going to use $\mathcal{F}$ to design our new ZKCSP protocol as follows. Informally, the server/seller will choose a random r and send to the client/buyer the value $y = \mathcal{F}_{f,\mathsf{H}}(s,r)$ and proves using a WI protocol that he knows inputs (s,r) such that $y = \mathcal{F}_{f,\mathsf{H}}(s,r)$. Note that if $f(s) = 1$ then $y = \texttt{SHA256}(r)$, otherwise $y = \mathsf{H}(r)$. Moreover, if the output of H "looks like" the output of $\texttt{SHA256}$, the client/buyer cannot tell at this point if the server actually knows a "good" s (i.e. $f(s) = 1$) or not. To detect if this is the case or not should be contingent to a payment by the buyer who therefore publishes the following transaction:

Transfer m Bitcoins to the party who presents a $\texttt{SHA256}$ preimage of y

If $f(s) = 1$ then the server/seller knows such a preimage (which is r), and can publish it to redeem the payment. Moreover, if we assume that finding a $\texttt{SHA256}$ preimage of $\mathsf{H}(r)$ is hard, then the seller cannot redeem payment when $f(s) \neq 1$.

More formally, let A denote the seller, B denote the buyer and T denote the blockchain:

Protocol 1

1. A on input s, chooses r at random in $\{0,1\}^{256}$ and computes $y = \mathcal{F}_{f,\mathsf{H}}(s,r)$
2. A sends y to B and the two parties engage in a WI proof that the seller knows r, s such that $y = \mathcal{F}_{f,\mathsf{H}}(s,r)$. If the proof fails, the buyer rejects and stops.
3. B posts a transaction to the Bitcoin blockchain to pay m Bitcoins to the party who presents x such that $\texttt{SHA256}(x) = y$
4. A presents z to T. If $\texttt{SHA256}(z) = y$ then T posts it and the seller redeems the Bitcoins, otherwise the Bitcoins are returned to the buyer.

We can prove the following[10]:

Theorem 1. *Assume that*

- $\texttt{SHA256}$ *and* H *are a claw-free pair*
- *the distributions* $\texttt{SHA256}(r)$ *and* $\mathsf{H}(r)$ *for* r *chosen at random in* $\{0,1\}^{256}$ *are computationally indistinguishable*

then Protocol 1 *is a secure ZKCSP protocol.*

[10] The assumptions underlying Theorem 1 are expressed in asymptotic terms but for sake of simplicity we are using concrete security parameters and functions used by Bitcoin (e.g. $\texttt{SHA256}$, with 256 bits output etc). It is easy to reframe the protocol description and the theorem using a security parameter

Proof. (Sketch).

Extraction: Let $\hat{A}$ be an efficient, possibly malicious seller. In step 2, $\hat{A}$ runs a ZK proof of knowledge of the values s, r which can therefore be extracted if the proof is successful. Assume for sake of contradiction that $f(s) \neq 1$ and $\hat{A}$ gets paid. By the correctness of the NIZK we know that since $f(s) \neq 1$, then $y = \mathsf{H}(r)$. In Step 4 $\hat{A}$ gets paid only if she produces z such that $y = \texttt{SHA256}(z)$. Therefore we have found a claw (r, z) for SHA256 and H, since $\texttt{SHA256}(z) = \mathsf{H}(r) = y$.

Zero-Knowledge is a consequence of the witness indistinguishability of the proof in step 2, and the computational indistinguishability of the output distributions of SHA256 and H. A bit more formally, For step 1, $Sim_{\hat{B}}$ will choose r, s at random and compute $y = \mathcal{F}_{f,\mathsf{H}}(s, r)$. Note that the message in step 1 is computationally indistinguishable from the message sent by the real A due to the computational indistinguishability of the output distributions of SHA256 and H. For step 2, $Sim_{\hat{B}}$ will just run a "real" proof that $y = \mathcal{F}_{f,\mathsf{H}}(s, r)$: note that due to witness indistinguishability, this proof is indistinguishable from a proof of a "correct" proof when the witness is such that $f(s) = 1$.

Letting B choose the CRS. Note that we only require the proof to be WI. If we were to use a QSP-based SNARK, such as Libsnark, then (as already pointed out in [29]) the verifier B can be allowed to select the CRS, provided the prover A performs some minimal correctness checks (described in detail in Section 3.3).

4.3 An alternative ZKCP construction

The idea behind our ZKCSP can be used to build an alternative ZKCP protocol. Recall (using the notation in Section 2.5) that in this case, Alice (the seller) wants to sell to Bob (the buyer) a string f_A that satisfies some verification procedure V_B.

The basic idea remains the same: Alice encrypts the string f_A with a key k (using any symmetric encryption scheme E, i.e. AES) and publishes $\hat{f} = E_k(f_A)$ and $y = \texttt{SHA256}(k)$. She then proves (using a WI proof) that

$$y = \mathcal{G}_{V_B,\hat{f},\mathsf{H}}(k) = \begin{cases} \texttt{SHA256}(k) \text{ if } V_B(D_k(\hat{f})) = 1 \\ \mathsf{H}(k) \text{ otherwise.} \end{cases} \tag{2}$$

Note that in this case WI is sufficient since at the end of the protocol, Bob does not know if Alice encrypted a valid string or garbage, and this guarantees that he learns no information about f_A. At the same time, he is guaranteed that if Alice presents a SHA256 preimage of y, then the encrypted string must be valid and he will be able to recover it. Again relying simply on WI, removes the need for a trusted party to generate the CRS, since Bob can be allowed to generate it, provided that Alice performs the minimal checks to guarantee WI (described in Section 3.3) and without having to resort to the heavy tests required by subversion-ZK.

4.4 A Protocol with private verification

In the protocol above we assumed a scenario in which anybody can verify that s is "correct" (i.e. $f(s) = 1$). There are however situations in which the buyer is the only one who can verify the correctness of s. In other words the buyer is only interested in s such that $f(k, s) = 1$ where k is a secret "key" held by the buyer. In this case we modify the protocol to have the parties jointly compute the following function

$$\mathcal{F}'_{f,\mathsf{H}}(k,s,r) = \begin{cases} \texttt{SHA256}(r) \text{ if } f(k,s)=1 \\ \mathsf{H}(r) \text{ otherwise.} \end{cases} \tag{3}$$

Because both buyer and seller want to keep k and s secret respectively, they will have to use a secure two-party computation protocol, such as Yao's garbled circuit [51] to compute $\mathcal{F}'$. It is important to use a two-party computation protocol which is secure against malicious players. The protocol is described below

Protocol 2

1. A on input s, chooses r at random in $\{0,1\}^{256}$
2. Using a 2-party computation protocol, secure against malicious players, A and B jointly compute $y = \mathcal{F}'_{f,\mathsf{H}}(k,s,r)$ where k is B's private input.
3. B posts a transaction to the Bitcoin blockchain to pay m Bitcoins to the party who presents x such that $\texttt{SHA256}(x) = y$
4. A presents z to T. If $\texttt{SHA256}(z) = y$ then T posts it and the seller redeems the Bitcoins, otherwise the Bitcoins are returned to the buyer.

Theorem 2. *Assume that*

- SHA256 *and* H *are a claw-free pair*
- *the distributions* $\texttt{SHA256}(r)$ *and* $\mathsf{H}(r)$ *for* r *chosen at random in* $\{0,1\}^{256}$ *are computationally indistinguishable*

then Protocol 2 *is a secure ZKCSP protocol.*

Proof. (Sketch).

Extraction: Let $\hat{A}$ be an efficient, possibly malicious seller. In step 2, $\hat{A}$ runs two-party computation protocol which is secure against a malicious adversary. Such protocols require the ability to extract the input during the simulation [36], so we use the simulator of the two-party protocol to extract r, s. Now the proof continues as in the proof of Theorem 1. Assume for sake of contradiction tha $f(k,s) \neq 1$ and $\hat{A}$ gets paid. By the correctness of the two-party computation protocol we know that since $f(k,s) \neq 1$, then $y = \mathsf{H}(r)$. In Step 4 $\hat{A}$ gets paid only if she produces z such that $y = \texttt{SHA256}(z)$. Therefore we have found a claw (r,z) for SHA256 and H, since $\texttt{SHA256}(z) = \mathsf{H}(r) = y$.

Zero-Knowledge is a consequence of the simulatability of the two-party protocol in step 2, and the computational indistinguishability of the output distributions of SHA256 and H. A bit more formally, For steps 1 and 2, $Sim_{\hat{B}}$ will choose r at random and compute $y = \texttt{SHA256}(r)$ and simulate the two-party computation with y as output. Now if A has a correct s then step 4 will be executed and the simulator will simulate it perfectly by releasing a SHA256 preimage of y. If A did not have a correct s, then step 4 is a message between A and T but is not posted to the blockchain and therefore does not belong to the view of $\hat{B}$ and the simulator does not have to simulate it.

5 Implementation

In this section we discuss our implementation work on: the attack against Maxwell's ZKCP; a proof of concept of our protocol for ZKCP for Services; a more efficient `SHA256` circuit implementation (used in Protocol 2). The code is available at [21,22]. All benchmarks in this section were evaluated on a Debian 3.16.39-1 x86_64 GNU/Linux Virtual Machine (virtual CPU and RAM respectively 2.4 GhZ and 3.5 GB).

5.1 Pay-to-Sudoku

Attack. We modified the Pay-to-Sudoku's code [18] in a way that allows a malicious buyer to learn information about the value of a cell of the Sudoku solution without paying for it. To do that we created a modified version of `libsnark` that implements the attack described in Section 3 (under "Setting all the pk equal to the identity except for one wire"). The malicious buyer can generate a CRS running this code and find out the exact value of a cell with probability at least 1/9 from the proof received by the seller. Note that the seller in [18] does not find out the CRS was generated maliciously and that we did not modify any code involving the Sudoku solution seller or the prover in libsnark.

Alternative Pay-to-Sudoku. We also implemented our alternative ZKCP protocol using only WI proofs (described in Section 4.3) for the case of Pay-to-Sudoku. In our protocol the prover runs a bit slower than the insecure original protocol due to the fact that the proof is run over a larger circuit (verification time is basically unchanged as to be expected in the case of QSP-based protocols). On the other hand, the cost of the expensive subversion-ZK CRS checks to the original Pay-to-Sudoku protocol dominates the overhead of the larger circuit in our protocol (which does not require such expensive checks). In particular, **our results suggest that the proving process would require more than an hour in total** (instead of a few seconds without the ZK-subversion checks). This time has been obtained by computing $t_P \cdot n_P$, where t_P is the experimental estimate for the the average time per pairing check (i.e. 4.50 ms) and n_P is the number of pairing checks for subversion-ZK in [28]. A lower bound on n_P is $7m$ where m is the number of constraints. The quantity m is sligthly greater than 115K for Pay-to-Sudoku. In these benchmarks we used curve ALT_BN128, the same originally used in Pay-to-Sudoku. Table 1 summarizes the performance comparison.

	ZKCSP for Sudoku with WI checks	**Pay-to-Sudoku with Subversion-ZK**
Key Generation	54 s	22s
Proof	10900 ms	> 1 hour (5500 ms without checks)
Verification	25 ms	24 ms

Table 1. Estimated Running Time for Contingent Payment for Sudoku *with checked CRS*

5.2 Proofs of Retrievability (PoR) over Bitcoin

As a proof of concept, we provide an implementation for a ZKCSP for Auditing of Proofs of Retrievability (PoR). Our implementation is based on the PoR scheme in [44] (See Appendix for details of the the scheme). In the context of PoR, a party delegates storage of her data to a server. A PoR scheme consists of an (efficient) protocol by which the delegator can verify at any time whether the server is still keeping her data intact. Our protocol allows the client to pay the server if such verification procedure succeeds. The PoR scheme in [44] can be instantiated both as privately and publicly verifiable (see appendix for details). For this application, the curve we used in `libsnark` was MNT6. Although less efficient than BN128 or ALT_BN128, this curve was one of the few ones which offered verification gadgets for pairings.

Private Verification. In this case the PoR scheme in [44] reduces to the verification of a (linearly homomorphic) MAC jointly by the server and the client. Here the PoR is successful if the server proves to the client that it knows $s = (m, t)$ such that $t = MAC_k(m)$ where k is the secret authentication key of the client.

We used Protocol 2 described in Section 4.4 where $f(k, s) = 1$ if and only if $s = (m, t)$ and $t = MAC_k(m)$. We implemented a two-party protocol for the computation of the associated function $\mathcal{F}'$ using the SCAPI library [26] following [48]. We used $\lambda = 128$ bits of computational security and $\rho = 80$ bits of statistical security. We chose a Carter-Wegman [49] style MAC, specifically the one in [34]. The circuit has 150441 gates and 151017 wires. The number of input wires for the two parties, seller and buyer, are respectively 416 and 160. The output of the circuit is 256 bits. See Table 2 for evaluation of running time and bandwidth.

	Bandwidth (KB)	**Time (ms)**
Garbler	38879	155
Evaluator	51	159

Table 2. Stats for Fair Auditing of Privately Verifiable PoR with Secure Two Party Computation.

Public Verification. In this case the PoR scheme in [44] reduces to the verification of a (linearly homomorphic) signature scheme, specifically the BLS scheme from [15]. More specifically the PoR is succesfull if the server proves to the client that it knows $s = (m, \sigma)$ such that $Ver(PK, m, \sigma) = 1$ where Ver is the verification algorithm of the BLS signature scheme, and PK is the public key of the client.

In this case we used Protocol 1 described in Section 4.2 where $f(s) = 1$ if and only if $s = (m, \sigma)$ and $Ver(PK, m, \sigma) = 1$. We implemented ZK-SNARK to enable the server to prove that she knows (s, r) such that $y = \mathcal{F}_{f,\mathsf{H}}(s, r)$. This proof was implemented in C++ using `libsnark` [11]. The function $\mathcal{F}$ was described in `libsnark` as set of constraints called Rank-One Constraint System (R1CS). Implementing the above $\mathcal{F}$ we obtained a R1CS system with 39409 constraints. In this setting we used $\lambda = 80$ bits of computational security. See Table 3 for evaluation of running time and bandwidth.

For

Generation of the CRS The timing results in the tables above refer to a Key Generation performed by a trusted party.

	Bandwidth	Time (ms)
Key Generation	pk: 41959 KB sk: 13 KB	14041
Proof	374 bytes	3287
Verification	—	37

Table 3. Stats for Fair Auditing of Publicly Verifiable PoR with SNARKs.

5.3 A More Efficient `SHA256` Circuit Implementation

SCAPI and other cryptographic libraries require the user to supply the circuit for the function that want to compute. Building a circuit file in this format is complex, and there is a library of such circuit files made available by researchers at Bristol University [45].

As part of the implementation in the proof of concept above, we constructed a new optimized reusable boolean circuit for `SHA256`. Our circuit may be of independent use for circuit-based MPC and FHE protocols that require `SHA256` computations.

To the best of our knowledge, the only other re-usable circuit implementation openly available for `SHA256` was developed the Bristol circuit. See Table 4 for a comparison of the circuit parameters between the Bristol circuit and ours. Our circuit compares favorably both with respect to the total number of gates and to the number of AND gates. The latter parameter is particularly important if one intends to use `SHA256` in Secure Multi-Party Computation. In fact, in modern MPC protocols the number of AND gates dominates the total evaluation cost thanks to a technique called Free-XOR [33] which evaluates XOR gates "for free". In the process of building our `SHA256` circuit we developed a library for semi-automated generation of optimized boolean circuits which we believe may be of independent interest. We stress that our contribution here is not the optimizations themselves as they were mostly straightforward from the SHA2 specification, but or contributions is the the optimized implementation of SHA2 in a boolean circuit format that can be reused by other cryptographic libraries and protocols.

	Bristol Circuit	Our Circuit
Total gates	236112	**116245**
AND gates	90825	**22272**
XOR gates	**42029**	91780
INV gates	103258	**2194**

Table 4. Number of gates in `SHA256` circuit implementations.

6 Acknowledgments

We thank Dario Fiore, Hugo Krawczyk, Arvind Narayanan, Pino Persiano, and Eran Tromer and the anonymous reviewers for useful discussions and advice.

Matteo Campanelli is supported by NSF Grant 1545759. Rosario Gennaro is supported by NSF Grant 1565403. Steven Goldfeder is supported by the NSF Graduate Research Fellowship under grant number DGE 1148900 and NSF award CNS-1651938.

References

1. B. Abdolmaleki, K. Baghery, H. Lipmaa, and M. Zajac. A subversion-resistant snark. Cryptology ePrint Archive, Report 2017/599, 2017. `http://eprint.iacr.org/2017/599`.
2. G. Andresen. Github: Proposal: open up IsStandard for P2SH transactions. `https://gist.github.com/gavinandresen/88be40c141bc67acb247`, 2017.
3. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via bitcoin deposits. In R. Böhme, M. Brenner, T. Moore, and M. Smith, editors, *FC 2014 Workshops*, volume 8438 of *LNCS*, pages 105–121. Springer, Mar. 2014.
4. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. On the malleability of bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*, pages 1–18. Springer, 2015.
5. N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures (extended abstract). In K. Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 591–606. Springer, May / June 1998.
6. W. Banasik, S. Dziembowski, and D. Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *European Symposium on Research in Computer Security*, pages 261–280. Springer, 2016.
7. M. Bellare, G. Fuchsbauer, and A. Scafuro. Nizks with an untrusted crs: security in the face of parameter subversion. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Part II*, pages 777–804. Springer, 2016.
8. E. Ben-Sasson, A. Chiesa, D. Genkin, S. Kfir, E. Tromer, M. S. L. Virza, and others external contributors. Libsnark, 2017. `https://github.com/scipr-lab/libsnark`.
9. E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *IEEE Security and Privacy Conference*, pages 287–304, 2015.
10. E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Aug. 2014.
11. E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 781–796, Berkeley, CA, USA, 2014. USENIX Association.
12. BitcoinWiki. Bitcoin transaction, 2016. `https://en.bitcoin.it/wiki/Transaction`.
13. BitcoinWiki. Zero knowledge contingent payment, 2016. `https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment`.
14. BitcoinWiki. Scripts, 2017. `https://en.bitcoin.it/wiki/Script`.
15. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.
16. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.
17. J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies, 2015.
18. S. Bowe. pay-to-sudoku, 2016. `https://github.com/zcash/pay-to-sudoku`.
19. S. Bowe, A. Gabizon, and M. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. 2016. `https://github.com/zcash/mpc/blob/master/whitepaper.pdf`.
20. V. Buterin et al. A next-generation smart contract and decentralized application platform, 2014.
21. M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo. An attack to pay-to-sudoku. `https://github.com/matteocam/pay-to-sudoku-attack`, 2017.
22. M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo. Zkcsp over bitcoin. `https://github.com/matteocam/zkcsp-over-bitcoin`, 2017.
23. R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In J. Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369. ACM, 1986.
24. G. G. Dagher, B. Bünz, J. Bonneau, J. Clark, and D. Boneh. Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 720–731. ACM, 2015.
25. G. Danezis, C. Fournet, J. Groth, and M. Kohlweiss. Square span programs with applications to succinct NIZK arguments. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 532–550. Springer, Dec. 2014.

26. Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell. Scapi: The secure computation application programming interface. *IACR Cryptology EPrint Archive*, 2012:629, 2012.
27. U. Feige and A. Shamir. Witness indistinguishable and witness hiding protocols. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 416–426, 1990.
28. G. Fuchsbauer. Subversion-zero-knowledge snarks. Cryptology ePrint Archive, Report 2017/587, 2017. `http://eprint.iacr.org/2017/587`.
29. R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, May 2013.
30. O. Goldreich and H. Krawczyk. On the composition of zero-knowledge proof systems. *SIAM J. Comput.*, 25(1):169–192, 1996.
31. O. Goldreich and Y. Oren. Definitions and properties of zero-knowledge proof systems. *J. Cryptology*, 7(1):1–32, 1994.
32. A. Juels and B. S. Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. Acm, 2007.
33. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. *Automata, Languages and Programming*, pages 486–498, 2008.
34. H. Krawczyk. Lfsr-based hashing and authentication. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839, pages 129–139. Springer, 1994.
35. A. Küpçü and A. Lysyanskaya. Usable optimistic fair exchange. In J. Pieprzyk, editor, *CT-RSA 2010*, volume 5985 of *LNCS*, pages 252–267. Springer, Mar. 2010.
36. Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 52–78. Springer, 2007.
37. G. Maxwell. Zero knowledge contingent payment, 2015. `https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment`.
38. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
39. A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.
40. B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.
41. A. D. Santis and G. Persiano. Zero-knowledge proofs of knowledge without interaction (extended abstract). In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 427–436, 1992.
42. E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.
43. H. Shacham and B. Waters. Compact proofs of retrievability. In J. Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 90–107. Springer, Dec. 2008.
44. H. Shacham and B. Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107. Springer, 2008.
45. S. Tillich and N. Smart. Circuits of basic functions suitable for mpc and fhe, 2016.
46. F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Euro Security and Privacy'17, 2017. To appear.
47. J. van Lint. Introduction to coding theory, 1992.
48. X. Wang, A. J. Malozemoff, and J. Katz. Faster secure two-party computation in the single-execution setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 399–424. Springer, 2017.
49. M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.
50. G. Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.
51. A. C. Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

A APPENDIX: The Shacham/Waters POR Scheme

A Proof of Retrievability (PoR) scheme involves a client C, who outsources some data, and a server S who is supposed to store them, in a way that he can prove to a verifier that he is actually storing the client's data. In [43], Shacham and Waters presented two compact proof of retrievability schemes, one with private and another with public verifiability. The first one is based on PRFs and secure in the standard model, the second one based on BLS signatures [16], secure in the Random Oracle Model. The framework is the same for both: an erasured coded file is divided into n blocks $m_1, \ldots, m_n \in \mathbb{Z}_p$, where p is a large prime. Intuitively, the fact that the file is erasured coded ensures that it is possible to decode even in presence of adversarial (or random) erasure (see [47] for further details about erasure codes).

Privately Verifiable POR Scheme: In order to authenticate each block m_i, the client C chooses a secret key which is composed by a random $\alpha \xleftarrow{\$} \mathbb{Z}_p$ and a PRF key k for a function f. Then, for each $i \in [n]$ she computes $\sigma_i := f_k(i) + \alpha m_i \ \in \mathbb{Z}_p$.

The pairs $\{(m_i, \sigma_i)\}_{i\in[n]}$ are then stored into the server and the proof of retrievability between the server and the verifier works as follows:

1. The verifier chooses a challenge set $I \subset [n]$, $\mid I \mid = \ell$ and some coefficients $\nu_1, \ldots \nu_\ell \in \mathbb{Z}_p$. The set $Q := \{(i, \nu_i)\}_{i\in[\ell]}$ is then sent to the server.
2. The server sends back a pair (σ, μ), where

$$\sigma \leftarrow \sum_{(i,\nu_i)\in Q} \nu_i \cdot \sigma_i \text{ and } \mu \leftarrow \sum_{(i,\nu_i)\in Q} \nu_i \cdot m_i.$$

3. The verifies checks whether the following holds

$$\sigma = \alpha \cdot \mu + \sum_{(i,\nu_i)\in Q} \nu_i \cdot f_k(i)$$

.

Note the here the secret key is necessary in order to run the verification.

Public Verifiable POR Scheme: Let $e : G \times G \to G$ be a bilinear map and let $\mathbb{Z}_p$ be the support of G . The client sets a secret key to be $x \leftarrow \mathbb{Z}_p$ and the public key to be $(v := g^x, u)$, where g, u are two generators of G. Then, for each $i \in [n]$ she computes $\sigma_i := \left[H(i)u^{m_i}\right]^x$. As before, the pairs $\{(m_i, \sigma_i)\}_{i\in[n]}$ are then stored into the server and the proof of retrievability between the server and the verifier works as follows:

1. The verifier chooses a challenge set $I \subset [n]$, $\mid I \mid = \ell$ and some coefficients $\nu_1, \ldots \nu_\ell \in \mathbb{Z}_p$. The set $Q := \{(i, \nu_i)\}_{i\in[\ell]}$ is then sent to the server.
2. The server sends back a pair (σ, μ), where

$$\sigma \leftarrow \prod_{(i,\nu_i)\in Q} \sigma_i^{\nu_i} \text{ and } \mu \leftarrow \sum_{(i,\nu_i)\in Q} \nu_i \cdot m_i.$$

3. The verifies checks whether the following holds:

$$e(\sigma, g) = e(\prod_{(i,\nu_i)\in Q} H(i)^{\nu_i} \cdot u^{\mu}, v)$$

.

Note that the secret key x is necessary in order to create the authenticators $\{\sigma_i\}$. On the other hand the public element v is sufficient to perform the verification.