Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts

Wacław Banasik, Stefan Dziembowski, and Daniel Malinowski(✉)

University of Warsaw, Warsaw, Poland
Daniel.Malinowski@crypto.edu.pl

Abstract. One of the most promising innovations offered by the cryptographic currencies (like Bitcoin) are the so-called *smart contracts*, which can be viewed as financial agreements between mutually distrusting participants. Their execution is enforced by the mechanics of the currency, and typically has monetary consequences for the parties. The rules of these contracts are written in the form of so-called "scripts", which are pieces of code in some "scripting language". Although smart contracts are believed to have a huge potential, for the moment they are not widely used in practice. In particular, most of Bitcoin miners allow only to post standard transactions (i.e.: those without the non-trivial scripts) on the blockchain. As a result, it is currently very hard to create non-trivial smart contracts in Bitcoin.

Motivated by this, we address the following question: "is it possible to create non-trivial efficient smart contracts using the standard transactions only?" We answer this question affirmatively, by constructing efficient Zero-Knowledge Contingent Payment protocol for a large class of NP-relations. This includes the relations for which efficient sigma protocols exist. In particular, our protocol can be used to sell a factorization (p, q) of an RSA modulus $n = pq$, which is an example that we implemented and tested its efficiency in practice.

As another example of the "smart contract without scripts" we show how our techniques can be used to implement the contract called "trading across chains".

1 Introduction

Cryptographic currencies (also dubbed the *cryptocurrencies*) are a very interesting concept that emerged in the last few years. The most prominent of them, and by far the largest one (in terms of capitalization), is Bitcoin, introduced in 2009 [32]. The main property of these currencies is that their security does not rely on any single trusted third party. The list of transactions in the system is written on a public *ledger* that is maintained jointly by the users. Another

This work was supported by the WELCOME/2010-4/2 grant founded within the framework of the EU Innovative Economy Operational Programme and by the Polish National Science Centre grant 2014/13/B/ST6/03540.

I. Askoxylakis et al. (Eds.): ESORICS 2016, Part II, LNCS 9879, pp. 261–280, 2016.
DOI: 10.1007/978-3-319-45741-3_14

reason why these currencies are so interesting is that they allow the users to perform much more than simple money transfers between each other. Namely, several cryptocurrencies, including the Bitcoin, implement an idea of the so-called *smart-contracts*. Such contracts can be viewed as distributed protocols executed between a number of parties. Typically, they have financial consequences, i.e., the users contribute money to them, and these funds are later distributed among the participants according to contract rules. Moreover, these contracts are "self-enforcing", which means that their execution is guaranteed by the rules of the underlying cryptocurrency. In particular, once a party enters into such a contract she cannot "change her mind" and withdraw her invested funds unless the contract specifically allows her to do so.

To be more specific, consider a contract called the *Zero Knowledge Contingent Payment* [16], which is an example on how Bitcoin contracts can provide a solution for the so-called *fair exchange problem* (see, e.g., [34]). It is executed between two parties: the Seller and the Buyer. The Buyer is looking for a value $x \in \{0,1\}^*$, that he does not know, but he is able to specify the conditions of x that make it valuable for him. Namely, he can describe a function $f : \{0,1\}^* \to \{\mathsf{true}, \mathsf{false}\}$ (in a form of a polynomial-time computer program, say), such that every x satisfying $f(x) = \mathsf{true}$, has a value ₿100 for him (here "₿" denotes Bitcoin currency unit). Obviously (assuming that $\mathrm{P} \neq \mathrm{NP}$), *finding* x such that $f(x) = \mathsf{true}$ is much harder than verifying that $f(x) = \mathsf{true}$ holds. Hence, in many cases it makes a lot of sense for the Buyer to pay for x. As an example: think of a Buyer that wants to buy a factorization p, q of an RSA modulus N. He would then define $f : \mathbb{N} \times \mathbb{N} \to \{\mathsf{true}, \mathsf{false}\}$ as $f(p, q) := \mathsf{true}$ iff $((p \cdot q = N) \wedge p \neq 1 \wedge q \neq 1)$.

Imagine now that the Buyer is approached by a Seller, who is claiming that he knows x such that $f(x) = \mathsf{true}$ and he is willing to sell it. If this happens over the Internet, and the parties do not trust each other then they face the following problem: shall the Seller first send x to the Buyer who later pays to him (after verifying that indeed $f(x)$), or the other way around: shall the Buyer first pay and get x from the Seller? Clearly in the first case a malicious Buyer can refuse to pay ₿100 to the Seller (after receiving x), and in the latter a malicious Seller may not send x to the Buyer (after receiving the payment). Is there a way to sell x in such a way that none of the parties can cheat the other one? Unfortunately, it turns out (see, e.g., [33]), that this fundamental problem, called the *fair exchange* cannot be in general solved without a trusted third party. This is exactly where the contracts come to play. Intuitively, thanks to this feature of the cryptocurrencies, the users can use the ledger as a trusted entity that allows them to perform the exchange x for ₿100 simultaneously. Technically (but still very informally), this is done by placing a contract C on the ledger that has the following semantics: *"The Buyer has to put aside* ₿100*. This money can be claimed by the Seller only by posting x such that $f(x) = \mathsf{true}$ on the ledger. If he does not do it within time t, then* ₿100 *goes back to the Buyer."* Now, everybody who observes the ledger can easily verify if the contract obligations were respected by the parties, and decide whether ₿100 should be now "transferred" from the Buyer to the Seller or not.

Another interesting example of a contract is so-called *trading across chains* [12] where users can exchange in a secure and fair way money between different cryptographic currencies. More advanced examples include, the *rapidly-adjusted micro-payments*, the *assurance contracts* [12], the multiparty lotteries [4,6], or general secure multiparty computation protocols [2,11,27]. Some experts predict that the smart contracts will revolutionize the digital economy. It is even envisioned that in the future these contracts may be used to maintain large *distributed autonomous corporations* that would operate without any trusted party control [22].

1.1 Contracts: From Theory to Practice

The above description ignores many technical details, and in particular it does not mention how the contracts are written. The transactions that are used in the contracts contain the so-called *scripts*. In Bitcoin the scripts are written in the so-called *Bitcoin script language* [13], which is not Turing-complete, and hence not every condition can be expressed in it. A serious obstacle when implementing the Bitcoin contracts in real life is that in practice it is currently very hard to post on the ledger a transaction corresponding to a non-trivial contract. Technically, to write a transaction on the ledger one broadcasts it over Bitcoin network and hopes that one of the miners (which are the entities that are maintaining the ledger) will include it into a new block that he mines. This gives the miners power to decide which transactions are included into the blockchain and which are not. Unfortunately, currently most of the miners do not include more complicated transactions into the blockchain. The reasons for this are: (1) such transactions tend to be longer than the "standard" ones, and space in the block is scarce, and (2) writing the transactions is tricky and error-prone, and most of the mining pool operators agreed to disallow them in order to prevent the users from loosing money. Technically deciding whether to accept a transaction or not is done by computing a boolean function `isStandard()` that evaluates to `true` only if the transaction is "standard", and otherwise it evaluates to `false`. The vast majority of the miners will include a transaction T in a new block only if `isStandard(`T`)`= `true` (more on this can be found, e.g., in [5], Chap. 5). Up to our knowledge, the only mining pool that currently accepts the non-standard transactions is *Eligius* that mines less than 1 % of blocks.

Another problem with running the smart contracts in Bitcoin is that the Bitcoin scripting language contains a feature, called the *transaction malleability*, that makes it tricky to implement several natural contracts (for more on this see the extended version of this paper [7], or, e.g., [3]). Although some techniques of dealing with this problem are known [3], they are often hard to use, since they make the contracts unnecessarily complicated (and make the transactions longer), and sometimes force the parties to invest more money than would normally be needed (by requiring them to put aside so-called *deposits*). One interesting new tool for dealing with this problem is the OP_CHECKLOCKTIMEVERIFY instruction [38] that was recently deployed.

After Bitcoin was deployed several other cryptocurrencies were proposed. The most interesting one from the point of view of the smart contracts, is *Ethereum*, which permits to use the Turing-complete scripts. The aforementioned problem of the high time consumption associated with the evaluation of the complicated scripts is solved in Ethereum in the following way. Each step of the computation of a script costs some small amount of money (the currency used for this is called *ether*), and the script evaluates as long as there are enough funds for this. Ethereum has recently been deployed in real life. It is, however, still a very young project and it is unclear how successful it will be in the real life. Moreover, as recently observed by Luu et al. [29] Ethereum may be susceptible to attacks where the adversary wastes miners' computational resources, which, in turn means that the miners might have incentives not to verify the correctness of the scripts. This, at least in theory, puts the whole Ethereum security model at risk.

Some of the other new cryptocurrencies go in the opposite direction by removing the possibility of having scripts at all. Sometimes this is a price for having additional interesting features in a currency. One example is the *Zerocash* [10], where the key new feature is the real anonymity (obtained by using the zero-knowledge techniques). Another, slightly different example is the *Lightning* system, which is a new proposal for micropayments constructed on top of the Bitcoin financial system, that also allows only standard transactions between the parties.

1.2 Our Contribution: Contracts Without Scripts

These observations lead to the following natural question: can we *efficiently* construct non-trivial contracts using only the standard transactions? In this paper we answer this affirmatively. We show (in Sect. 3.2) a general technique for efficiently solving the Zero-Knowledge Contingent Payment problem *using only standard transactions* for any f such that the corresponding language $\{x : f(x) = \mathsf{true}\}$ has an efficient zero-knowledge proof of knowledge of a special (but very broad) form, that, in particular, includes the sigma-protocols (see, e.g., [20]). We define this class of protocols in Sect. 3.3, but for a moment let us only say that it includes many natural languages. As an example we show an efficient protocol for selling a factorization of an RSA modulus, which is a problem that we already discussed at the beginning of this section. We implemented our protocol and confirmed its efficiency (see Sect. 3.4). In our construction we do not rely on any costly cryptographic mechanisms such as the generic secure multiparty computation protocols, or the generic zero-knowledge schemes. Instead, we use the standard and simple cut-and-choose technique. Our techniques can also be used to solve, in a similar way, the "trading across chains" problem. Because of the lack of space this is shown in the extended version of this paper [7].

Our protocols are proven secure in the random oracle model, and are based on standard cryptographic assumptions, an assumption that time-lock encryption of [37] is secure, plus one additional assumption about the strong unforgeability of the Elliptic Curve DSA (ECDSA) signatures used in Bitcoin. We describe this assumption in more detail in Sect. 2. Our protocols have an exponentially small probability of error (i.e.: the probability that the adversary cheats), assuming that we are allowed to use so-called *multisig* transactions, i.e., transactions that can be spent by providing signatures with respect to k public keys (out of $n \geq k$ possible public keys). Currently such transactions are considered standard for $n \leq 15$. We note that if one does not want to use such transactions, then our solution also works, but the error probability is inversely proportional to the running time of the parties.

Related work. As already mentioned, the *Zero-Knowledge Contingent Payment* protocol has been described before in [16] and recently implemented [31] for selling a proof of a sudoku solution. When viewed abstractly, our construction is a bit similar to the one of [16]. There are some important differences, though. Firstly, the protocol of [16] uses some non-standard scripts. Secondly, it is vulnerable to the "malleability attacks": the *refund transaction* depends on an identifier of the *txn* transaction, and becomes meaningless if *txn* is mauled. Finally, the protocol of [16] uses generic zero knowledge protocols, or can be used only for very simple problems (like selling the sudoku solution), while we rely on much simpler and more efficient methods (in particular: the *cut-and-choose* technique).

2 Preliminaries

Definitions. We will sometimes model the hash functions as *random oracles*, see [9]. A *signature scheme* consists of a *key generation algorithm* $\mathsf{SignGen}$, a *signing algorithm* Sign, and a *verification algorithm* Vrfy. For a formal definition of a signature scheme see [26], or the extended version of this paper [7]. The standard security notion for signatures is the *existential unforgeability under a chosen message attack*. In this paper we need a stronger security definition, namely the *strong* existential unforgeability under a chosen message attack. This is formally defined in [1,18]. Essentially, the definition is as follows. Consider the standard chosen-message attack during which the adversary interacts with a signing oracle that knows some secret key sk. We say that $\mathcal{A}$ *mauls a signature* if he is able to produce an output $(\hat{z}, \hat{\sigma})$ such that $\hat{\sigma}$ is a valid signature on $\hat{z}$ with respect to the public key pk (that corresponds to sk), and $\hat{\sigma}$ has not been sent to $\mathcal{A}$ before. A signature scheme is *existentially strongly unforgeable under a chosen message attack* (or: *non-malleable*) if for any polynomial-time adversary the probability that he mauls a signature is negligible.

We will use (*public key* and *private key*) *encryption schemes*, defined in a standard way (see [26] or [7].) We say that a public-key encryption scheme is *additively homomorphic* if for every valid public key pk and private key sk the set of valid messages for pk is an additive group $(\mathbb{H}_{pk}, +)$. Moreover, we require that there exists an operation $\otimes : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \cup \{\perp\}$, such that for every valid (pk, sk) and every pair $z_0, z_1 \in \mathbb{H}_{pk}$ we have that $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(z_0) \otimes \mathsf{Enc}_{pk}(z_1)) = z_0 + z_1$ (where Enc and Dec are the encryption and decryption algorithms, respectively).

Our protocols also rely on the *time-lock commitment schemes* [17,37] (for the definition of the standard commitment schemes see, e.g., [26], or [7]). Informally, $(\mathsf{Commit}, \mathsf{Open})$ is a *time-locked commitment* if it is a standard commitment scheme, except that the receiver can open the commitment by himself (even if the sender is not cooperating). Such *forced opening* requires a significant computational effort. Moreover it is required that this process cannot be parallelized. Every time-lock commitment comes with two parameters: τ_0 and τ_1 (with $\tau_0 \leq \tau_1$), where τ_0 denotes the time (in seconds, say) that everybody, including very powerful adversaries, needs to force open the commitment, and τ_1 denotes time needed by the honest users to force open the commitment. We will call such a commitment scheme (τ_0, τ_1)-secure. Of course, this is not a formal mathematical definition (as it refers to "real time"), but for the purpose of this paper we can stay on this informal level. Later, in Sect. 3.4 we assume that $\tau_1 = 10 \cdot \tau_0$, but this choice is slightly arbitrary, and for real practical applications one would need to perform a more careful analysis of what is the reasonable ratio between τ_0 and τ_1 that one can assume.

For a description of the area of *zero-knowledge* the reader may consult, e.g., [24] (a brief introduction also appears in [7]). In our paper we actually need a stronger notion, namely the *zero-knowledge proofs of knowledge* [8]. Such proofs are defined only if L is in NP, and hence for every $x \in L$ there exists an *NP-witness* w that serves as a proof that $x \in L$. We assume that P knows x and require that the prover not only proves that $x \in L$, but also convinces the verifier that he knows the corresponding witness w. Defining formally the property of a prover "knowing" some value is a bit tricky, and we do not do it here (see, e.g., [24] for such a definition). Very informally, it is usually defined as follows: for every (possibly malicious) prover P^* there exists a polynomial-time machine, called the *knowledge extractor*, that can interact with P^* (possibly even rewinding it), and at the end it outputs x. The definition that we use here is more restrictive. First, suppose without loss of generality, that the last two messages in the protocol are: a challenge c sent by the verifier to the prover, and provers response r. We require (cf. Sect. 3.3) that the extractor extracts the witness after being given transcripts of two accepting executions that are identical except that that the challenge messages are different (and the response messages may also be different). This class of protocols includes our protocol for selling the factorization of the RSA modulus. It is also similar to the sigma-protocols (see, e.g., [20]), except that it may have more rounds than 3, but on the other hand we require that the zero-knowledge property holds also against the malicious verifier. Note that some sigma-protocols, including the Schnorr protocol, are conjectured to be secure also in this case. Observe also that we can easily get rid of the "honest verifier" assumption by requiring the verifier to make his message equal to a hash of some message (chosen by him) [21]. Hence, our method can be used also to efficiently "sell" a witness of any relation for which an efficient sigma-protocol exists.

Instantiations. As explained in the introduction, Bitcoin uses an *Elliptic Curve Digital Signature Algorithm (ECDSA)* [19,25], which is a variant of the *Digital Signature Algorithm (DSA)*. More concretely, it uses the *Secp256k1* curve [14], but to be able to state our theorems in an asymptotic way we will be more general and define our protocol over arbitrary elliptic curve. The description of this algorithm appears in [7].

As it turns out, these signatures are *not* strongly unforgeable: if (r, s) is a valid signature on some message z, then also $(r, -s \bmod p)$ (where p is the order over which the elliptic curve $\mathbb{G}$ is defined) is a valid signature with respect the same public key (see, e.g., [7] for more on this). In order to make our signature scheme strongly-unforgeable we follow the guidelines from [39]. Namely, we assume that the only "legal" signatures have a form (r, s) such that $s \leq (p-1)/2$. To this end, we simply assume that, whenever our protocol gets as input an ECDSA signature (r, s) with $s > (p-1)/2$, it converts it to one with $s \leq (p-1)/2$ by computing $s := -s \bmod p$. An ECDSA scheme with only "legal" signatures being the ones with $s \leq (p-1)/2$ will be called a *positive ECDSA*.

We can now informally state our strong unforgeability assumption as follows: "*The positive ECDSA defined over Secp256k1 is strongly unforgeable under chosen-message attack*" (or equivalently: the only way to maul the signatures defined over Secp256k1 is to negate the s). Note that this statement is informal, and in order to formalize it we would need to express it in an asymptotic way. See [7] for more on this, and on the general issue of the malleability of Bitcoin transactions.

We will use the additively-homomorphic public key encryption scheme introduced by Pascal Paillier [35]. Below, we describe only the properties of this scheme that are needed in this work. For more details the reader can consult, e.g., [35]. The public key pk of this encryption scheme contains a modulus $n = p \cdot q$, where p and q are large distinct random primes of the same length. The Paillier encryption scheme is homomorphic over $(\mathbb{Z}_n, +)$. It is semantically secure under the *Decisional composite residuosity assumption* [35]. In the sequel we will assume that $(\mathsf{AddHomGen}, \mathsf{AddHomEnc}, \mathsf{AddHomDec})$ is a Paillier encryption scheme. The elements on which we will perform the addition operations will be the exponents in the elliptic curve group of the ECDSA scheme. Hence, we need $\mathbb{Z}_n$ to be larger than $\mathbb{G}$, and, for the reasons that will become clear later, it will be convenient to have $n \gg |\mathbb{G}|$. We therefore assume that on input 1^λ the algorithm $\mathsf{AddHomGen}$ produces as output (pk, sk) such that the corresponding group $\mathbb{Z}_n$ satisfies $n > 2 \cdot |\mathbb{G}|^4$.

We use very standard commitment schemes that are based on the hash functions, and are secure in the random oracle model. Let H be a hash function. In order to commit to $x \in \{0,1\}^*$ the committer chooses random $r \in \{0,1\}^\lambda$ (where 1^λ is the security parameter) and produces as output $\mathsf{Commit}(x) = H(x||r)$. In order to open the commitment it is enough to reveal (x, r). The fact that the scheme is binding follows from the collision-resistance of H. The hiding property follows from the fact that we model H as the random oracle (and hence $H(x||r)$ does not reveal any information about x).

We use the classic timed commitments of [37]. In order to commit to a message $x \in \{0,1\}^\ell$ (for some ℓ) the committer chooses an RSA modulus n, i.e., he selects two random primes p and q of length λ (where 1^λ is the security parameter) and sets $n = pq$. He then computes $\varphi(n) = (p-1)(q-1)$. Let t be some parameter. The committer takes random $y \in Z_n^*$ and computes $z := y^{2^t} \bmod n$. Since he knows $\varphi(n)$ he can compute it efficiently by first computing $e = 2^t \bmod \varphi(n)$ (doing this using the standard square-and-multiply algorithm takes $\log_2 t$ squaring modulo n), and then letting $z := y^e \bmod n$. Finally, he computes $H(z)$ and outputs y and $H(z) \oplus x$, where $H : \mathbb{Z}_n^* \to \{0,1\}^\ell$ is a hash function. On the other hand, it is conjectured [37] that an adversary, who does not know $\varphi(n)$ needs to perform t squarings to compute z (and hence to compute x). Also, no practical methods of parallelizing the problem of computing z is known. It is also easy to see that this algorithm is a commitment in a standard sense, i.e., if the committer is cooperating with the receiver then he can open the commitment efficiently (by sending (p, q) to the receiver). To set the parameter t let c be the number of squarings that the honest receiver can do in one second. We then let $t = \tau_1 \cdot c$ (where τ_1 is the parameter of the timed commitment scheme).

A short description of the Bitcoin transaction syntax. We now briefly describe the syntax of the Bitcoin transactions. A more complete description can be found, e.g., in [5,7,15]. Since we do not use the non-standard transactions we will provide a simplified description that ignores this feature of Bitcoin. The users in Bitcoin are identified by their public keys in the ECDSA signature scheme $(\mathsf{SignGen}, \mathsf{Sign}, \mathsf{Vrfy})$. Each such a key pk is called an *address*. In the simplest case transaction T simply sends some amount ₿x (where x can be smaller than one) from an address pk_0 (called an *input* of T) to an address pk_1 (called the *output* of T). The amount ₿x will also be called the *value of* T. Transaction T must contain a pointer to another transaction T' that appeared earlier on the ledger and has value at least ₿x, and whose destination is pk_0. We say that T *redeems* T'. Transaction T is valid only if T' has not been redeemed by some other transaction before. Hence, in the simplest case a transaction contains a following tuple $[T] := (\texttt{TXid}(T'), \texttt{value} : ₿x, \texttt{from} : pk_0, \texttt{to} : pk_1)$, where $\texttt{TXid}(T')$ denotes the *identifier of* T' (we will define it in a moment), and $[T]$ is called a *simplified transaction* T. Of course, in order for $[T]$ to have any meaning it needs to be signed with the private key sk_0 corresponding to pk_0. Hence, the complete transaction T has a form $([T], \mathsf{Sign}_{sk_0}([T]))$, and is valid if all the conditions described above hold, and the signature on $[T]$ is valid with respect to pk_0. The $\texttt{TXid}(T)$ is defined simply as a SHA256 hash of $([T], \mathsf{Sign}_{sk_0}([T])))$.

Another standard type of the transactions are the so-called *multisig* transactions. In this case $[T]$ has a form $(\texttt{TXid}(T'), \texttt{value} : ₿x, \texttt{from} : pk_0, \texttt{to}$ "k-$\texttt{out-of-}n$" $: pk_1, \ldots, pk_n)$ where $n \leq 15$. It is signed by pk_0. It can be spent by a transaction T'' that is signed by k signatures with respect to k different public keys from the set $pk_1, \ldots, pk_n$. More precisely the transaction

T'' has to have a form $([T''], \sigma_{i_1}, \ldots, \sigma_{i_k})$, where $1 \leq i_1 < \cdots < i_k \leq n$ and for every $1 \leq j \leq k$ holds $\mathsf{Vrfy}_{pk_{i_j}}([T''], \sigma_{i_j}) = \mathsf{ok}$.

3 The Protocols

Our model. We will consider two-party protocols, executed between a Buyer B and a Seller S. If a party is malicious then she may not follow the protocol (in other words: we consider the *active* security settings). The parties are connected by a secure (i.e. secret and authenticated) channel. Such a channel can be easily obtained using the standard techniques, provided that the parties know each others public keys. Observe that in order to do any financial transfers in Bitcoin they anyway need to know each other keys (let (sk_B, pk_B) be the ECDSA key pair of the Buyer, and let (sk_S, pk_S) the key pair of the Seller), and the participating parties can use the same key pairs for establishing the secure channel between each other. How exactly these public keys pk_B and pk_S are exchanged is beyond the scope of this paper.

The security definition. We now outline a construction of our protocol in which the Seller sells to the Buyer x such that $f(x) = \mathsf{true}$ (for some public $f : \{0,1\}^* \rightarrow \{\mathsf{true}, \mathsf{false}\}$). We assume that the "price" of x is ₿d, and that, before an execution of the protocol starts, there is some unspent transaction T_0 on the blockchain whose value is ₿d, and whose output is pk_B (i.e.: it can be spent by the Buyer). The parties initially share the following common input: a security parameter 1^λ, a price ₿d for the secret x, parameters $a, b \in \mathbb{N}$ such that $a > b$, an elliptic curve group $(\mathbb{G}, \mathcal{O}, g, +)$ for an ECDSA signature scheme, such that $\lceil \log_2 |\mathbb{G}| \rceil = \lambda$, and parameters (τ_0, τ_1). We say that the $\mathsf{SellWitness}_f$ protocol is ϵ*-secure* if the following properties hold: (1) except with probability $\epsilon + \mu(\lambda)$ (where μ is negligible), if an honest Buyer loses his funds then he learns x' s.t. $f(x') = \mathsf{true}$, (2) except with negligible probability, if an honest Seller does not get Buyer's funds then the Buyer learns no information about x. We construct a protocol $\mathsf{SellWitness}_f$ (for a large class of functions f) in Sect. 3.3. First, however, we give an outline of our construction. The necessary ingredients are defined and constructed in Sects. 3.1 and 3.2.

Outline of the construction. Our protocol consists of several stages. The main idea can be described as follows (we start with describing an "idealized" protocol and then we show how to modify it to make it efficient and practical). Imagine that the parties first create, in a distributed way, an ECDSA key pair (sk, pk) such that the private key sk is secret-shared between the parties, and the public key pk is known to both of them. Then, the Buyer prepares a transaction T_1 that sends the output of T_0 to the public key pk. Obviously for a moment the Buyer has to keep T_1 private, as posting T_1 on the ledger would put his money at risk (as spending money from T_1 requires cooperation of the Seller). He now creates a simplified transaction[1] $[T_2]$ that redeems T_1 and sends the output to the public key pk_S of the Seller. Then, the parties jointly sign $[T_2]$ with the shared private key sk in such a way that the signature $\sigma = \mathsf{Sign}_{sk}([T_2])$ is known only to the Seller. Note that this is possible without revealing T_1 to the Seller, as the only thing that is needed from T_1 is its transaction identifier, which happens to be equal to the hash $H(T_1)$ of T_1 (in the random oracle model $H(T_1)$ clearly reveals no information about T_1).

Let us now briefly analyze the situation after these steps are executed: the Buyer knows T_1, and the Seller knows T_2 that spends T_1 (but she does not know T_1, so for a moment she cannot make any use of T_2). The key idea now is: the Seller will make a commitment to the signature σ in such a way that opening this commitment will automatically reveal x (and she will convince the Buyer that the commitment was formed in this way). Now the Buyer can post T_1 on the ledger, and wait until the Seller redeems it. The only way in which she can do it, is to publish σ (here we use the assumption that the signatures are strongly unforgeable), so the Buyer can be sure that he learns x.

This construction is similar to the one described in [16]. Unfortunately, in practice there are several problems with it. Firstly, there is no way for the Buyer to "force" the Seller to publish σ, and hence the Buyer's money can be locked forever in T_1. We solve this problem using the time-locked commitments. The Seller has to commit with such a commitment to her private share of sk, so that it can be unlocked by the Buyer after some time. In this way he can get his money back by signing a transaction T_2' that redeems T_1 and sends the money to his key pk_B. As described in Sect. 1, an alternative solution is to use the OP_CHECKLOCKTIMEVERIFY instruction. We describe this solution in the extended version of this paper [7].

Secondly, the currently-known protocols for distributed signing with the ECDSA signatures are rather complicated and involve costly generic zero-knowledge techniques [30] (see also [23]). Also, the generic zero-knowledge would need to be used to prove that the timed commitment above is indeed a commitment to Seller's share in sk.

Our solution to this problem is to use the standard technique, called *cut-and-choose* (see, e.g., [28]). Informally, the idea here is to perform a number a of independent executions of a protocol. Then the Buyer tells the Seller to "uncover" $a - b$ (for some parameter $b < a$) of them, by opening all her commitments related to these executions. It is easy to see that, if all the opened commitments were correct, then most probably a significant fraction of the remaining b ("non-uncovered") executions will also be correct. Since some executions may still be incorrect, we will thus create T_1 as a multisig transaction (so it can be spent with less than b signatures). This is done in Sects. 3.1 and 3.2. Thirdly, we need to describe how to create the commitment to σ in the last step that requires proving that "opening this commitment will automatically reveal x". We do it as follows: we require that the Seller commits to $F(\sigma)$ (where F is some hash function),

[1] Recall (cf. Sect. 2) that a "simplified transaction" means a transaction without a signature.

1. The parties run a times the SharedKGen protocol to generate secret-shared signing keys.
2. The Buyer selects b of these keys and uses GenMsg_T to produce transactions T_1 and T_2.
3. The parties run the USG protocol to sign T_2 using all a shared keys and the Seller generates commitments. Then the Buyer checks the Seller on the unselected $a - b$ executions.
   – The single signing iteration is performed using the KSignGen procedure.
4. Using the Zero Knowledge protocol (and again the cut-and-choose technique) the Seller proves that by revealing any signature the Buyers will extract the witness x from it.
5. The Buyer broadcasts T_1. Then the Seller uses the signatures to broadcast T_2 and the Buyer can extract the witness x (or solve the timed commitment to get his funds back).

Fig. 1. The outline of the $\mathsf{SellWitness}_f$ protocol and the subprotocols.

and then we use again the cut-and-choose technique (on the elements of $F(\sigma)$) to prove that if the whole $F(\sigma)$ is opened then x is revealed. Technically, this is done by showing that revealing $F(\sigma)$ opens commitments to messages from a zero-knowledge proof of knowledge of x. For the details see Sect. 3.3. The outline of the $\mathsf{SellWitness}_f$ protocol and the subprotocols is presented on Fig. 1.

3.1 The Two-Party ECDSA Key Generation Protocol

The first ingredient of our scheme is a protocol in which two parties, the Seller and the Buyer, generate a (public key, private key) key pair for the ECDSA signatures, in such a way that the secret key is secret-shared between the Seller and the Buyer. To be more precise, fix an elliptic curve $(\mathbb{G}, \mathcal{O}, g, +)$ constructed over a field $\mathbb{Z}_p$ and recall that the secret key in the ECDSA signatures is a private integer $d \in \mathbb{Z}_{|\mathbb{G}|}$. We construct a two-party protocol, that we call SharedKGen, in which both parties take as input a security parameter 1^λ and at the end they both know an ECDSA public key $pk = d \cdot g$ (where d is secret), and additionally the Seller knows $d_S \in \mathbb{Z}_{|\mathbb{G}|}$ and the Buyer knows $d_B \in \mathbb{Z}_{|\mathbb{G}|}$ such that $d_S \cdot d_B = d \pmod{|\mathbb{G}|}$ is a secret-sharing. The protocol is very similar to the classic actively-secure key generation protocols for the discrete log signatures [36]. Because of the lack of space it is presented in the extended version of this paper [7].

3.2 The Unique Signature Generation Protocol

After the parties generate a key pairs $(sk^1, pk^1), \ldots, (sk^a, pk^a)$ using the SharedKGen protocol, they perform an additional procedure, called *unique signature generation (*USG*) protocol*, whose goal is to sign a message $z \in \{0,1\}^*$ with respect to these keys. The message z is chosen by the Buyer and may depend on the public keys that were generated in the SharedKGen phase, and on the Buyer's private randomness. During the execution of the USG protocol $a - b$ private keys are "uncovered" (here $b < a$ is some parameter), i.e., they are reconstructed by the parties. At the end of the execution they are discarded and the output of the protocol depends only on the key pairs whose private keys were not uncovered. Let $(\hat{sk}_1, \hat{pk}_1), \ldots, (\hat{sk}_b, \hat{pk}_b)$ denote these key pairs. Each $\hat{pk}_i$ is known to both parties, and each $\hat{sk}_i$ remains secret and is shared between the parties (as a pair $(\hat{d}_S^i, \hat{d}_B^i)$ of shares). Moreover the Seller knows the ECDSA signatures $\hat{\sigma}_1, \ldots, \hat{\sigma}_b$ on z with respect to $\hat{pk}_1, \ldots, \hat{pk}_b$ (respectively). The Buyer does not know these signatures, but we require that the Seller is committed (again: using COM) to each $F(\hat{\sigma}_i)$, where F is a hash function (modeled as a random oracle). Let $\Gamma_1, \ldots, \Gamma_b$ denote the commitments created this way. Finally, we want the Buyer to be able to "force open" the values $\hat{d}_S^1, \ldots, \hat{d}_S^b$ after some time τ_1, so that he can reconstruct the private keys $\hat{sk}_1, \ldots, \hat{sk}_b$ and sign any message that he wants using these keys. This is achieved using a (τ_0, τ_1)-secure time-locked commitment scheme $\mathsf{TLCOM} = (\mathsf{TLCommit}, \mathsf{TLForceOpen})$. Let $\Phi_1, \ldots, \Phi_b$ denote the timed-commitments that were created this way.

To explain informally our security requirements, first let us say what are the goals of a malicious Seller. One obvious goal is to produce a signature on some message $z^* \neq z$ (with respect to some $\hat{pk}_i$). A more subtle (and more specific to our applications) goal for the Seller is to learn some signature σ_i^* on z (with respect to one of $\hat{pk}_1, \ldots, \hat{pk}_b$) other than $\hat{\sigma}_1, \ldots, \hat{\sigma}_b$. Finally, she could try to time-commit to some value other than $\hat{d}_S^i$ (so that, after time τ_1 passes, the Buyer cannot reconstruct $\hat{sk}_i$). Formally, we say that the malicious Seller S^* *breaks the key i (for $i = 1, \ldots, b$)* if the Buyer did not abort the protocol and one of the following holds:

- after the execution of the protocol S^* produces as output $(\hat{\sigma}_i^*, \hat{z}_i)$ such that $\hat{\sigma}_i^*$ is a valid signature on $\hat{z}_i \neq z$ with respect to $\hat{pk}_i$,
- after the execution of the protocol S^* produces as output $\hat{\sigma}_i^*$ such that $\hat{\sigma}_i^*$ is a valid signature on z with respect to $\hat{pk}_i$, and S^* opens the commitment Γ_i to a value different than $F(\hat{\sigma}_i^*)$,
- the value d_B^{i*} that results from forced opening of Φ_i is such that $\hat{d}_S^i \cdot d_B^{i*} \neq \hat{d}^i$.

Now, consider a malicious Buyer. Informally, his goal is to learn any valid signature on z with respect to any key $\hat{pk}_1, \ldots, \hat{pk}_b$. If he does not succeed in this, then another goal that he could try to achieve is to learn at least one of the $F(\hat{\sigma}_i)$'s. Recall also that the secrets of the Seller are time-locked. Hence after time τ_0 the Buyer can easily "break" the protocol, and our definition has to take care of it. Formally, we say that a malicious Buyer B^* *wins* if the Seller did not abort the protocol and before time τ_0 one of the following holds:

- the B^* produces as output a signature on z^* (either $z^* = z$ or $z^* \neq z$) that is valid with respect to one of the $\hat{pk}_i$'s,
- the B^* learns some information about one of the $F(\hat{\sigma}_i)$'s.

We say that a USG *protocol is $(\epsilon, \hat{b})$-secure* if (a) for every polynomial-time malicious Seller the probability that she breaks at least $\hat{b}$ keys is at most $\epsilon + \mu(\lambda)$, where μ is negligible, and (b) for every polynomial-time malicious Buyer the probability that he wins is negligible.

The implementation of the USG protocol. Our USG protocol is depicted on Fig. 2. We assume that before it is executed the parties run the $\mathsf{SharedKGen}$

procedure a times (on input 1^λ). We denote these executions as $\mathsf{SharedKGen}^i(1^\lambda)$ for $i = 1, \dots, a$. As a result of each execution $\mathsf{SharedKGen}^i$, both parties learn the public keys pk^i and they secret-share the corresponding secret keys sk^i (let (d^i_S, d^i_B) be the respective shares).

The USG protocol uses as a subroutine the protocol $\mathsf{KSignGen}$ from Fig. 3. This protocol allows the parties to sign a message z using the secret key that is secret shared $d = d_S \cdot d_B$. First they jointly create signing randomness K. Then the Seller creates a new key in the Paillier encryption scheme and sends the encryption of his share d_S of the signing key d to the Buyer. The Buyer calculates the encryption of the unfinished signature (using the homomorphic properties of the Paillier cryptosystem) and sends it to the Seller. Then the Seller decrypts it and completes the signature σ. At the end the Seller commits to $F(\sigma)$ and creates a timed commitment to d_S. We now have the following lemma, its proof appears in [7].

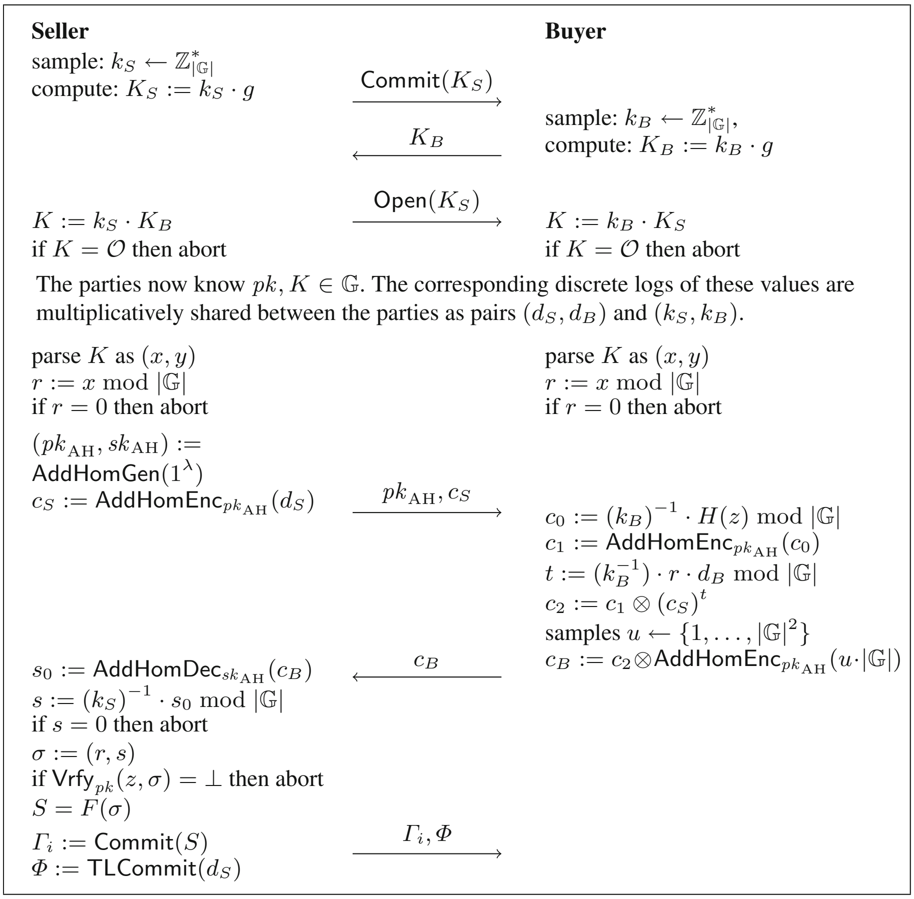
Lemma 1. *Suppose Paillier encryption is semantically secure,* COM *and* TLCOM *are secure commitment schemes, and the ECDSA scheme used in the construction of the* USG *is Strongly Unforgeable signature scheme. Then the* USG *protocol constructed on Fig. 2 is* $(\epsilon, \hat{b})$*-secure for* $\epsilon = (b/a)^{\hat{b}}$.

1. The Buyer chooses a random subset $\mathcal{J} \subset \{1, \dots, a\}$, such that $|\mathcal{J}| = a - b$. Let $\{j_1, \dots, j_b\}$ denote the set $\{1, \dots, a\} \setminus \mathcal{J}$.
2. The Buyer chooses a message z to be signed and sends it to the Seller.
3. For $i = 1$ to a the parties execute the $\mathsf{KSignGen}(1^\lambda)$ procedure depicted on Fig. 3. As a result of each such execution, the Seller is committed to $S^i = F(\sigma^i)$ and timed-committed to d^i_S.
4. The Buyer sends $\mathcal{J}$ to the Seller.
5. For every $j \in \mathcal{J}$ the Seller opens the commitments to S^j and d^j_S, and sends σ^j, k^j_S and sk^j_{AH} to the Buyer.
6. The Buyer aborts if any of the commitments did not open correctly. Otherwise he verifies if the following holds (for every $j \in \mathcal{J}$): (a) $\mathsf{Vrfy}_{pk^j}(z, \sigma^j) = \mathsf{ok}$, (b) $F(\sigma^j) = S^j$, (c) $d^j_S \cdot d^j_B \cdot g = pk^j$, and (d) $\mathsf{Dec}_{sk^j_{\mathrm{AH}}}(c^j_S) = d^j_S$,
7. If the verification fails then the Buyer aborts. If he did not abort then the parties use as output the values that were not open in Step 5. More precisely, the parties set $(\hat{sk}_i, \hat{pk}_i, \hat{\sigma}_i) := (sk^{j_i}, pk^{j_i}, \sigma^{j_i})$.

Fig. 2. The USG protocol.

3.3 The Construction of the $\mathsf{SellWitness}_f$ Protocol

In this section we show how to use the USG protocol to construct the $\mathsf{SellWitness}_f$ protocol (defined in Sect. 3). Our assumption is that f has a zero-knowledge proof of knowledge protocol, that we denote $\mathcal{F}$, in which the Seller can prove that she knows an x such that $f(x) = \mathsf{true}$. Additionally $\mathcal{F}$ consist of two phases: $\mathsf{Setup}_{\mathcal{F}}$ and $\mathsf{Challenge}_{\mathcal{F}}$. Let the values $A_{\mathcal{F}}$ and $B_{\mathcal{F}}$ denote the views of the Seller and the Buyer (respectively) after executing



Seller		**Buyer**
sample: $k_S \leftarrow \mathbb{Z}^*_{\lvert\mathbb{G}\rvert}$ compute: $K_S := k_S \cdot g$	$\xrightarrow{\mathsf{Commit}(K_S)}$	
	$\xleftarrow{K_B}$	sample: $k_B \leftarrow \mathbb{Z}^*_{\lvert\mathbb{G}\rvert}$, compute: $K_B := k_B \cdot g$
$K := k_S \cdot K_B$ if $K = \mathcal{O}$ then abort	$\xrightarrow{\mathsf{Open}(K_S)}$	$K := k_B \cdot K_S$ if $K = \mathcal{O}$ then abort

The parties now know $pk, K \in \mathbb{G}$. The corresponding discrete logs of these values are multiplicatively shared between the parties as pairs (d_S, d_B) and (k_S, k_B).

parse K as (x, y) $r := x \bmod \lvert\mathbb{G}\rvert$ if $r = 0$ then abort		parse K as (x, y) $r := x \bmod \lvert\mathbb{G}\rvert$ if $r = 0$ then abort
$(pk_{\mathrm{AH}}, sk_{\mathrm{AH}}) :=$ $\mathsf{AddHomGen}(1^\lambda)$ $c_S := \mathsf{AddHomEnc}_{pk_{\mathrm{AH}}}(d_S)$	$\xrightarrow{pk_{\mathrm{AH}}, c_S}$	$c_0 := (k_B)^{-1} \cdot H(z) \bmod \lvert\mathbb{G}\rvert$ $c_1 := \mathsf{AddHomEnc}_{pk_{\mathrm{AH}}}(c_0)$ $t := (k_B^{-1}) \cdot r \cdot d_B \bmod \lvert\mathbb{G}\rvert$ $c_2 := c_1 \otimes (c_S)^t$ samples $u \leftarrow \{1, \ldots, \lvert\mathbb{G}\rvert^2\}$ $c_B := c_2 \otimes \mathsf{AddHomEnc}_{pk_{\mathrm{AH}}}(u \cdot \lvert\mathbb{G}\rvert)$
$s_0 := \mathsf{AddHomDec}_{sk_{\mathrm{AH}}}(c_B)$ $s := (k_S)^{-1} \cdot s_0 \bmod \lvert\mathbb{G}\rvert$ if $s = 0$ then abort $\sigma := (r, s)$ if $\mathsf{Vrfy}_{pk}(z, \sigma) = \perp$ then abort $S = F(\sigma)$	$\xleftarrow{c_B}$	
$\Gamma_i := \mathsf{Commit}(S)$ $\Phi := \mathsf{TLCommit}(d_S)$	$\xrightarrow{\Gamma_i, \Phi}$	

Fig. 3. The $\mathsf{KSignGen}(1^\lambda)$ procedure. Recall that $\mathbb{G}$ is an elliptic curve group for ECDSA, and ($\mathsf{AddHomGen}, \mathsf{AddHomEnc}, \mathsf{AddHomDec}$) is a Paillier encryption scheme which is additively homomorphic over $\mathbb{Z}_n$, where $n > 2 \cdot \lvert\mathbb{G}\rvert^4$.

the $\mathsf{Setup}_{\mathcal{F}}$ phase. In the $\mathsf{Challenge}_{\mathcal{F}}$ phase the Buyer generates a challenge message $c_{\mathcal{F}} = \mathsf{GenChallenge}_{\mathcal{F}}(B_{\mathcal{F}})$ and sends it to the Seller. Then the Seller calculates the response $r_{\mathcal{F}} = \mathsf{GenResponse}_{\mathcal{F}}(x, A_{\mathcal{F}}, c_{\mathcal{F}})$ and sends it to the Buyer. At the end the Buyer accepts according to the output of the function $\mathsf{VerifyResponse}_{\mathcal{F}}(B_{\mathcal{F}}, c_{\mathcal{F}}, r_{\mathcal{F}}) \in \{\mathsf{true}, \mathsf{false}\}$. The fact that $\mathcal{F}$ is a proof of knowledge is formalized as follows: we require that there is also a function $\mathsf{Extract}_{\mathcal{F}}$ s.t. $\mathsf{Extract}_{\mathcal{F}}(B_{\mathcal{F}}, c^1_{\mathcal{F}}, r^1_{\mathcal{F}}, c^2_{\mathcal{F}}, r^2_{\mathcal{F}}) = x'$ and $f(x') = \mathsf{true}$ if only $\mathsf{VerifyResponse}_{\mathcal{F}}$ $(B_{\mathcal{F}}, c^i_{\mathcal{F}}, r^i_{\mathcal{F}}) = \mathsf{true}$ for $i = 1, 2$ and $c^1_{\mathcal{F}} \neq c^2_{\mathcal{F}}$. That means that the witness x' can be computed from the correct answers to two different challenges. We also assume that from the point of view of the Seller the challenge $c_{\mathcal{F}}$ is chosen uniformly from the set $X_{A_{\mathcal{F}}}$. Without loss of generality we also assume that $X_{A_{\mathcal{F}}} = \{0, 1\}$.

The parties use the USG protocol, so we have to describe how the Buyer produces the message z to be signed. Given the public keys $\hat{pk}_1, \ldots, \hat{pk}_b$ the Buyer first creates a transaction T_1 that takes ₿d from his funds and sends them to a multisig escrow "b-$\texttt{out-of-}$$(2b-1)$" using public keys $\hat{pk}_1, \ldots, \hat{pk}_b$ and $b-1$ times his own public key pk_B. The Buyer does not broadcast T_1 yet. Then he creates a transaction T_2 that spends the transaction T_1 and sends all the funds (₿d minus fee) to the public key pk_S owned by the Seller. The simplified transaction $z := [T_2]$ is the message that the parties later sign. We call this procedure GenMsg_T. We assume that each S^i from the USG protocol is divided into 2λ parts $S^{i,1}, \ldots, S^{i,2\lambda}$ each of size λ. Additionally we assume that each part $S^{i,j}$ is committed separately. To explain the idea behind our protocol assume for simplicity that $b = 1$. Recall that at the end of the USG protocol the Buyer knows the transaction T_1 that sends his funds to the key secret-shared between the Seller and the Buyer. Both parties know the transaction T_2 that is redeeming the transaction T_1 and sends the money to the Seller. The Seller knows the signature σ on T_2, but she cannot use T_2 yet, because the Buyer did not broadcast T_1. When the Buyer learns σ then he will be able to learn the secret random values $S^1, \ldots, S^{2\lambda}$ to which the Seller is committed. Additionally after some (long) time the Buyer will learn the secret key needed to redeem T_1 when only he force-opens the time-locked puzzle hiding d_S.

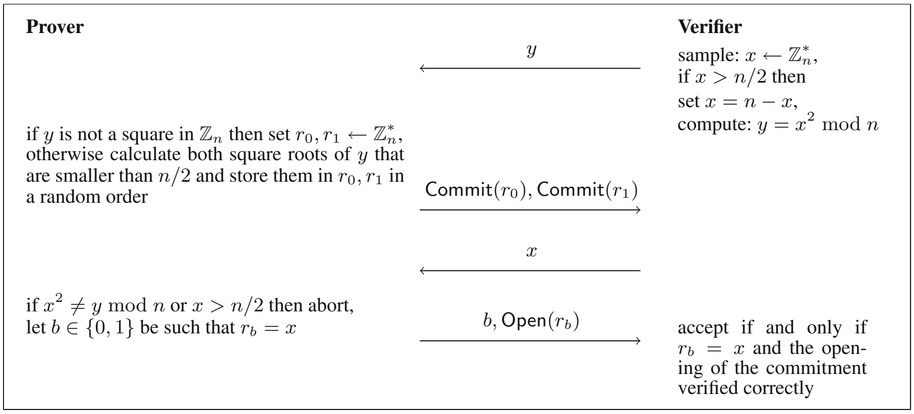
Now the Seller and the Buyer will use cut-and-choose technique again. They run 2λ times the first part $\mathsf{Setup}_{\mathcal{F}}$ of the zero knowledge proof of knowledge $\mathcal{F}$ of the x satisfying f. Each time the Seller calculates the responses r_0^i and r_1^i to the challenges $c = 0$ and $c = 1$. The Seller encrypts r_0^i and r_1^i using the same key S^i to get γ_0^i and γ_1^i and she commits to each ciphertext. Then the Buyer selects λ indices $j_1, \ldots, j_\lambda$ and challenges the Seller on them using $c_1, \ldots, c_\lambda \in \{0, 1\}$. The Seller opens commitments to $S^{j_1}, \ldots, S^{j_\lambda}$ and to $\gamma_{c_1}^{j_1}, \ldots, \gamma_{c_\lambda}^{j_\lambda}$ (the Seller opens only one of $\gamma_0^{j_k}, \gamma_1^{j_k}$) and the Buyer uses secrets S^{j_k} to decrypt $\gamma_{c_k}^{j_k}$ and verify the response. If the Buyer verifies everything without an error, then the Seller opens the commitments to γ_0^k and γ_1^k (but not S^k) for $k \neq j_1, \ldots, j_\lambda$.

Now the Buyer broadcasts the transaction T_1. The Seller can spend it by revealing σ — in that case the Buyer can compute S^k, decrypt γ_0^k and γ_1^k to learn responses r_0^k and r_1^k and from them extract the value x. And if the Seller does nothing then after some time the Buyer will solve his time-locked puzzle, learn the secret key and take his funds back. The $\mathsf{SellWitness}_f$ protocol is depicted on Fig. 4. We have the following lemma, its proof appears in [7].

Lemma 2. *Suppose Paillier encryption and symmetric encryption are semantically secure,* COM *and* TLCOM *are secure commitment schemes, and the ECDSA scheme used in the construction of the* USG *is Strongly Unforgeable signature scheme. Assume additionally that there is a zero knowledge proof* $\mathcal{F}$ *of knowledge of* x *s.t.* $f(x) = \mathsf{true}$ *of the required form. Then the* $\mathsf{SellWitness}_f$ *constructed on Fig. 4 is* ϵ*-secure for* $\epsilon = \left(\frac{b}{a}\right)^b$.

1. The parties execute the USG protocol using the provided parameters. The Buyer will generate transaction T_2 to be signed as defined earlier in the procedure GenMsg_T.
2. For $i = 1$ to b:
   a) For $j = 1$ to 2λ: the parties execute the $\mathsf{Setup}_{\mathcal{F}}^{i,j}$ phase and the Seller and the Buyer learns $A_{\mathcal{F}}^{i,j}$ and $B_{\mathcal{F}}^{i,j}$ respectively.
   b) For $j = 1$ to 2λ: the Seller calculates the two challenges (in random order) that can be chosen by the Buyer $c_1^{i,j}$ and $c_2^{i,j}$. Then she calculates the responses $r_k^{i,j} = \mathsf{GenResponse}_{\mathcal{F}}(x, A_{\mathcal{F}}^{i,j}, c_k^{i,j})$ for $k = 1, 2$.
   c) For $j = 1$ to 2λ: The Seller uses the secret $S^{i,j}$ as a key in the symmetric cypher and encrypts $\gamma_k^{i,j} = \mathsf{Enc}_{S^{i,j}}(c_k^{i,j}, r_k^{i,j})$ for $k = 1, 2$. Then she commits to $\gamma_k^{i,j}$ for $k = 1, 2$.
   d) The Buyer chooses random subset $\mathcal{J}^i \subset \{1, \dots, 2\lambda\}$ of size λ. Then he sends to the Seller $(j, c_B^{i,j} := \mathsf{GenChallenge}_{\mathcal{F}}(B_{\mathcal{F}}^{i,j}))$ for $j \in \mathcal{J}^i$.
   e) For $j \in \mathcal{J}^i$: the Seller opens her commitment to $S^{i,j}$ and checks that $c_B^{i,j} = c_k^{i,j}$ for $k = 1$ or $k = 2$. She opens the commitments to $\gamma_k^{i,j}$ for only this k.
   f) For $j \notin \mathcal{J}^i$: the Seller opens her commitments to $\gamma_k^{i,j}$ for $k = 1, 2$.
   g) The Buyer verifies all the commitments.
   h) For $j \in \mathcal{J}^i$: the Buyer decrypts $(c^{i,j}, r^{i,j}) = \mathsf{Dec}_{S^{i,j}}(\gamma_k^{i,j})$. Then he checks that $c^{i,j} = c_B^{i,j}$ and $\mathsf{VerifyResponse}_{\mathcal{F}}(B_{\mathcal{F}}^{i,j}, c_B^{i,j}, r^{i,j}) = \mathsf{true}$.
3. The Buyer broadcasts T_1 and the parties wait until it becomes final.
4. The Seller broadcasts T_2 using the signatures $\hat{\sigma}_1, \dots, \hat{\sigma}_b$ to get her payment.
5. The Buyer uses signatures $\hat{\sigma}_i$ to calculate secrets $S^{i,j}$. Then he decrypts all the values $\gamma^{i,j}$ to get all the challenges and responses $c_k^{i,j}, r_k^{i,j}$. At the end using any pair of responses he calculates $x' = \mathsf{Extract}_{\mathcal{F}}(B_{\mathcal{F}}^{i,j}, c_1^{i,j}, r_1^{i,j}, c_2^{i,j}, r_2^{i,j})$.
6. If the Seller do not redeem the Buyer's transaction then the Buyer force-opens time-locked puzzles Φ_i and uses any of the opened values d_S^i to get his funds back.

Fig. 4. The $\mathsf{SellWitness}_f$ protocol.



Prover		**Verifier**
	$\xleftarrow{y}$	sample: $x \leftarrow \mathbb{Z}_n^*$, if $x > n/2$ then set $x = n - x$, compute: $y = x^2 \bmod n$
if y is not a square in $\mathbb{Z}_n$ then set $r_0, r_1 \leftarrow \mathbb{Z}_n^*$, otherwise calculate both square roots of y that are smaller than $n/2$ and store them in r_0, r_1 in a random order	$\xrightarrow{\mathsf{Commit}(r_0), \mathsf{Commit}(r_1)}$	
	$\xleftarrow{x}$	
if $x^2 \neq y \bmod n$ or $x > n/2$ then abort, let $b \in \{0, 1\}$ be such that $r_b = x$	$\xrightarrow{b, \mathsf{Open}(r_b)}$	accept if and only if $r_b = x$ and the opening of the commitment verified correctly

Fig. 5. The $\mathsf{ZKFactorization}(n)$ protocol

3.4 Protocol for Selling a Factorization of an RSA Modulus

In this section we use the SellWitness protocol to construct the protocol for selling a factorization of an RSA modulus. To do it, we introduce the ZKFactorization protocol depicted on Fig. 5 — a zero knowledge proof of knowledge of the factorization of the RSA modulus. We now have the following lemma, whose proof appears in [7].

Lemma 3. *Assume that the commitment scheme is hash based and we model the hash function as a programmable oracle. Then the protocol* ZKFactorization *depicted on Fig. 5 is a zero knowledge proof of knowledge of the factorization of the RSA modulus.*

Implementation of the protocol for selling a factorization of an RSA modulus. We have created a prototype implementation of the protocol for selling a factorization of an RSA modulus. The main part of the protocol is written in C++, it is using the Crypto++ library for cryptographic functions. The Bitcoin related functionality is written in Java using the *bitocinj* library. The communication between C++ and Java is operated by *Apache Thrift*. The implementation is only a proof of concept but we were able to verify the feasibility and efficiency of the protocol. The current version of the protocol can be found on github.com/SellWitness/ZKFactorization. When using the ZKFactorization protocol in the SellWitness protocol we were able to simplify the main protocol a little. In the ZKFactorization protocol the Seller sends the commitments to the square roots of y but now it is not necessary because we do similar step in the SellWitness protocol. This is why the only messages exchanged between the parties before the Buyer sends the challenge are: first the Buyer sends $y^{i,j}$, then the Seller calculates the square roots $r_0^{i,j}, r_1^{i,j}$ of y, encrypts them $\gamma_k^{i,j} = Enc_{S^{i,j}}(r_k^{i,j})$ and commits to both $r_k^{i,j}$. In the implementation we use the following parameters: $a = 512$, $b = 8$ and $\lambda = 1024$. We use $b = 8$ because it means "b-out-of-$(2b-1)$" multisig transactions, and this kind of multisig transaction are standard in Bitcoin (for greater b they would be non-standard). We set $\lambda = 1024$, so the

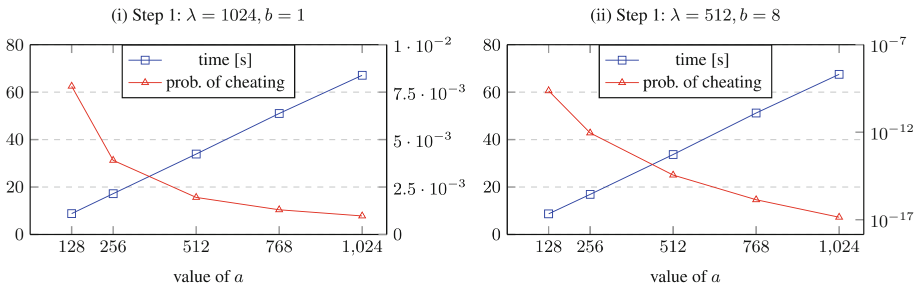


Fig. 6. The running time of the Step 1 and the probability that the Seller successfully cheats the Buyer in the Step 1 of the SellWitness protocol for the following fixed parameters: (i) $\lambda = 1024$ and $b = 1$ (i.e. using only standard single-signature transactions), and (ii) $\lambda = 512$ and $b = 8$ (i.e. using multi-signature transactions with the greatest parameters that are standard in Bitcoin) and different values of a. The running time of Step 1 is proportional to a and does not depend on other parameters. Using greater b gives much better security.

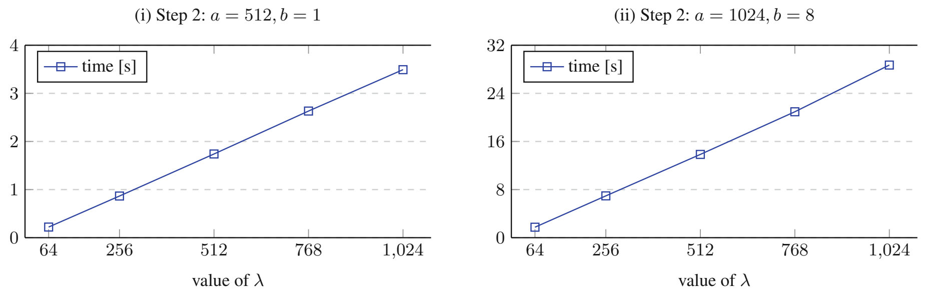


Fig. 7. The running time of the Step 2 of the SellWitness protocol for the following fixed parameter: (i) $a = 512$ and $b = 1$ (i.e. using only standard single-signature transactions), and (ii) $a = 1024$, and $b = 8$ (i.e. using multi-signature transactions with the greatest parameters that are standard in Bitcoin) and different values of λ. The running time of Step 2 is proportional to $b \cdot \lambda$ and does not depend on a. The probability of successfully cheating (by either the Buyer or the Seller) in step 2 is negligible in λ.

ZKFactorization protocol is executed $b \cdot 2\lambda = 8 \cdot 2048$ times. Fortunately this phase does not require any costly public key cryptography operations and therefore it is still very efficient. We set $a = 512$ and $b = 8$, and hence the probability of cheating is at most $(b/a)^b = 2^{-48}$. The running time of our protocol (i.e. the time until the Buyer broadcasts T_1) for this configuration (and primes of size about 512 bits each) is about 1 min — the running time of the USG protocol is about 33 s and Step 2 in the $\mathsf{SellWitness}_f$ protocol takes about 28 s. The numbers are an average over 10 runs of the algorithm using a single thread on a standard personal computer. We note that the running time could be improved by using multiple threads. Additional measurements are presented on Figs. 6 and 7.

We run our protocol on a single machine, and local testing blockchain (testnet-box) and hence posting on blockchain, and communication between the parties was almost immediate (our current implementation takes 12 rounds, and the total communication size is about 60 MB). However, since we use the time-lock commitment schemes we need a conservative estimate on how much time would the execution of our protocol take on real blockchain, and when the parties are running in different physical locations. As in our protocol the parties have to wait for two transactions to be included into the blockchain, we have to assume that the whole protocol may take up to two hours[2]. Taking into account time needed to post messages on the blockchain the running our protocol takes on average 2 h, we have to have at least $\tau_0 = 5$ h, so τ_1 should be set to 50 h. Our tests has shown that an honest user (on an standard personal computer) can compute about 2^{19} squares (modulo n of size $\lambda = 1024$ bits) per second. That is why in our protocol we set the hardness of the timed commitment to $t = 2^{37}$.

[2] It takes on average 10 min for a transaction to be included into the blockchain but the users are advised to wait for 6 blocks ($\approx$1 h) on top of the transaction.

References

1. An, J.H., Dodis, Y., Rabin, T.: On the security of joint signature and encryption. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 83–107. Springer, Heidelberg (2002)
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Fair two-party computations via bitcoin deposits. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) FC 2014 Workshops. LNCS, vol. 8438, pp. 105–121. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44774-1_8. http://dx.doi.org/10.1007/978-3-662-44774-1_8. ISBN: 978-3-662-44773-4
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: On the malleability of bitcoin transactions. In: Brenner, M., Christin, N., Johnson, B., Rohloff, K. (eds.) FC 2015 Workshops. LNCS, vol. 8976, pp. 1–18. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48051-9. http://dx.doi.org/10.1007/978-3-662-48051-9. ISBN: 978-3-662-48050-2
4. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 443–458. IEEE Computer Society Press, Berkeley (2014). doi:10.1109/SP.2014.35
5. Antonopoulos, A.M.: Mastering Bitcoin: Unlocking Digital Crypto-Currencies, 1st edn. O'Reilly Media, Inc., Sebastopol (2014). ISBN: 1449374042, 9781449374044
6. Back, A., Bentov, I.: Note on fair coin toss via Bitcoin. CoRR abs/1402.3698 (2014). http://arxiv.org/abs/1402.3698
7. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts. Cryptology ePrint Archive (2016). http://eprint.iacr.org/2016/451.pdf
8. Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 390–420. Springer, Heidelberg (1993)
9. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In: Ashby, V. (ed.) ACM CCS 1993, pp. 62–73. ACM Press, Fairfax (1993)
10. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society Press, Berkeley (2014). doi:10.1109/SP.2014.36
11. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44381-1_24
12. Bitcoin Wiki: Constract
13. Bitcoin Wiki: Script
14. Bitcoin Wiki: Secp256k1
15. Bitcoin Wiki: Transaction
16. Bitcoin Wiki: Zero Knowledge Contingent Payment
17. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 236–254. Springer, Heidelberg (2000)
18. Boneh, D., Shen, E., Waters, B.: Strongly unforgeable signatures based on computational diffie-hellman. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 229–240. Springer, Heidelberg (2006)
19. Brown, D.R.L.: Standards for Efficient Cryptography SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0 (2010)

20. Damgard, I.: On Sigma-Protocols (2015). http://www.cs.au.dk/~ivan/Sigma.pdf
21. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987)
22. Filippi, P.D.: Tomorrow's apps will come from brilliant (and risky) Bitcoin code. Wired magazine
23. Goldfeder, S., Gennario, R., Kalodner, H., Bonneau, J., Felten, E.W., Kroll, J.A., Narayanan, A.: Securing bitcoin wallets via a new DSA/ECDSA threshold signature scheme (manuscript, 2015)
24. Goldreich, O.: Foundations of Cryptography, vol. 1. Cambridge University Press, New York (2006). ISBN: 0521035368
25. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). Int. J. Inf. Secur. **1**(1), 36–63 (2001)
26. Katz, J., Lindell, Y.: Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series). Chapman & Hall/CRC, Boca Raton (2007). ISBN: 1584885513
27. Kumaresan, R., Moran, T., Bentov, I.: How to use bitcoin to play decentralized poker. In: Ray, I., Li, N., Kruegel, C. (eds.) Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, pp. 195–206. ACM, 12–16 October 2015. doi:10.1145/2810103.2813712. http://doi.acm.org/10.1145/2810103.2813712. ISBN: 978-1-4503-3832-5
28. Lindell, Y., Pinkas, B.: Secure two-party computation via cut-and-choose oblivious transfer. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 329–346. Springer, Heidelberg (2011)
29. Luu, L., Teutsch, J., Kulkarni, R., Saxena, P.: Demystifying incentives in the consensus computer. In: ACM CCS 2015, pp. 706–719. ACM Press (2015)
30. MacKenzie, P., Reiter, M.K.: Two-party generation of DSA signatures. English. Int. J. Inf. Secur. **2**(3–4), 218–239 (2004). doi:10.1007/s10207-004-0041-0. http://dx.doi.org/10.1007/s10207-004-0041-0. ISSN: 1615-5262
31. Maxwell, G.: The first successful Zero-Knowledge Contingent Payment (2016)
32. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2009). http://bitcoin.org/bitcoin.pdf
33. Pagnia, H., Gartner, F.C.: On the impossibility of fair exchange without a trusted third party. Technical report Darmstadt University of Technology (1999)
34. Pagnia, H., Vogt, H., Gärtner, F.C.: Fair Exchange. Comput. J. **46**(1), 55–75 (2003). doi:10.1093/comjnl/46.1.55. http://dx.doi.org/10.1093/comjnl/46.1.55
35. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
36. Pedersen, T.P.: A Threshold cryptosystem without a trusted party (Extended Abstract) (Rump Session). In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 522–526. Springer, Heidelberg (1991)
37. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release Crypto. Technical report Cambridge, MA, USA (1996)
38. Todd, P.: OP_CHECKLOCKTIMEVERIFY. Bitcoin Improvement Proposal 0062. https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki
39. Wuille, P.: Bitcoin Improvement Proposal 062: Dealing with malleability