Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts

Wacław Banasik, Stefan Dziembowski, and Daniel Malinowski

University of Warsaw

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 [36]. 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. Without going much into the details: the main property of the ledger is that it is *write-only*, which prevents the *double-spending* attacks (once a transaction appears on the ledger it is irreversible). Very informally, the ledger can therefore be viewed as a distributed replacement of a trusted third party (like a bank). Since it is public, everybody who observes it can examine the complete history of transactions and decide how much money is held on each account.[1] Another 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 Nick Szabo [44] 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. Hence, the contracts are binding in a similar way as the legal documents are, the main difference being that in the former case the execution of a contract is enforced automatically, while in the latter case the parties have to rely on the judicial system.

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,

[1] The accounts are identified by the public keys, and hence they cannot be automatically linked to individual users. This weak form of anonymity is also called "pseudonymity".

e.g., [38]). 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 P≠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 then 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., [37]), 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 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, 10, 31]. 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 [25].

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. The reason why Bitcoin was designed in this way is that otherwise it could be the case that a contract would require a very long computation, and hence its status, i.e. whether it was fulfilled or not, would not be immediately clear (recall that it is undecidable whether a program written in a Turing-complete language terminates or not). 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. To explain why it is the case we need to say a bit more about Bitcoin mechanics. The transaction ledger has a form of a chain of blocks (hence the name: the *blockchain*). Each block contains a list of new transactions and a pointer to the previous block. It is computationally moderately hard to extend the chain by a new block (this "moderate computational hardness" is obtained via a technique called the

proofs of work [23]), and hence it is also hard to delete a transaction from the blockchain (especially if it is contained in a block that is already followed by some other blocks in the chain) as it would require to produce an alternative chain without this transaction. The process of finding a new block is called *mining*, and the users that work on extending the chain are called the *miners*. They receive financial rewards (in bitcoins) for mining each new block. In order to reduce the variance of their income the miners typically operate in coalitions called the *mining pools*.

Technically, to write a transaction on the ledger one broadcasts it over Bitcoin network and hopes that one of the miners 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[2], 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 non-standard transactions are also called *strange*). 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], Chapter 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. We write more about it in Sect. 2. 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 [45] that was recently deployed in Bitcoin. See Sect. 4.3 for more information about this instruction.

After Bitcoin was deployed several other cryptocurrencies were proposed. The most interesting one from the point of view of the smart contracts, is *Ethereum* (`www.ethereum.org`), 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. [33] 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* [9], where the key new feature is the real anonymity (obtained by using the zero-knowledge techniques). Another, slightly different example is the *Lightning* system (`www.lightning.network`), 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. 4.2)

[2] Currently it is limited to 1MB, and there is a disagreement in the Bitcoin community whether to extend it or not (see [11])

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., [22]). We define this class of protocols in Sect. 4.3, 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. 4.4). 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. We also outline how to solve in a similar way the "trading across chains" problem (this is done in Sect. 4.5).

Our protocols are proven secure in the random oracle model, and are based on standard cryptographic assumptions, an assumption that time-lock encryption of [41] 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. 3. For a moment let us only briefly say that ECDSA signatures are malleable in the sense that given a valid (message, signature) pair (m, σ) it is possible to efficiently construct another pair $(m, g(\sigma))$ that is valid with respect to the same public key, and such that $\sigma \neq g(\sigma)$ (here g is some function defined over the set of valid signatures). Currently only one function g like this is known. Informally, our assumption states that no more functions like this exist. In Sect. 3 we informally explain why we believe that this is the case. We leave reducing this assumption to more standard assumptions as an interesting open problem. 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 [35] 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". To see why it is the case look at the section entitled "*But what if they're just anti-social and don't redeem the txn?*" in [16]: 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). The existing protocols for *trading across chains* [12] suffer from similar problems.

2 Preliminaries

Definitions.

Signature schemes We will sometimes model the hash functions as *random oracles*, see [8]. A *signature scheme* consists of a *key generation algorithm* $\mathsf{SignGen}$, a *signing algorithm* Sign, and a *verification algorithm* Vrfy. The algorithm $\mathsf{SignGen}$ takes as input a security parameter 1^λ, and produces as output a key pair $(sk, pk) \in (\{0,1\}^*)^2$. The *signing algorithm* takes as input sk, a message $z \in \{0,1\}^*$ and produces as output a *signature* $\sigma = \mathsf{Sign}_{sk}(z) \in \{0,1\}^*$. The *verification algorithm* Vrfy takes as input pk, z, and σ and produces as output $\mathsf{Vrfy}_{pk}(z, \sigma) \in \{\perp, \mathsf{ok}\}$. We say that σ *is a valid signature on* z *with respect to* pk if $\mathsf{Vrfy}_{pk}(z, \sigma) = \mathsf{ok}$. We require that always $\mathsf{Vrfy}_{pk}(z, \sigma) = \mathsf{ok}$, as long as $\sigma = \mathsf{Sign}_{sk}(z)$ and $(sk, pk) = \mathsf{SignGen}(1^\lambda)$ (i.e.: a correctly computed signature is always valid).

To define security of a signature scheme consider the following game played by a polynomial time adversary $\mathcal{A}$. First, a key pair is sampled as $(pk, sk) := \mathsf{SignGen}(1^\lambda)$ and the adversary is given pk. He then chooses a sequence $(z_1, \ldots, z_k)$ of messages, and he learns $\sigma_i := \mathsf{Sign}_{sk}(z_i)$ for each z_i. He does it *adaptively*, i.e. he chooses each z_i *after* learning $\sigma_1, \ldots, \sigma_{i-1}$. Finally, $\mathcal{A}$ produces as output a pair $(\hat{z}, \hat{\sigma})$. We say that $\mathcal{A}$ *forges a signature* if $\hat{z} \notin \{1, \ldots, z_k\}$ and $\hat{\sigma}$ is a valid signature on $\hat{z}$ with respect to pk. We say that $(\mathsf{SignGen}, \mathsf{Sign}, \mathsf{Vrfy})$ *is existentially unforgeable under a chosen message attack* if for any polynomial-time $\mathcal{A}$ the probability that $\mathcal{A}$ forges a signature is negligible in λ. 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]. Consider the same game as before. 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, i.e. $\hat{\sigma} \notin \{\sigma_1, \ldots, \sigma_k\}$. 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.

Homomorphic encryption schemes. We will use (*public key* and *private key*) *encryption schemes*, defined in a standard way (see [30] or Appendix A). 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 $(\mathbb{H}_{pk}, +)$ (note that this requires specifying how the group operation "+" works, but in our case this will always be clear from the context). Moreover, we require that there exists an operation $\otimes : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$, 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).

Time-lock commitment schemes. The definition of the standard commitment schemes is moved to Appendix A. Our protocols also rely on the *time-lock commitment schemes* [41, 17]. 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. In other words: no matter how many machines are available to the receiver, he always has to spend some significant amount of time in order to force open the commitment. Of course the machines differ in speed, and hence, a powerful receiver can speed up the forced opening process. However, as long as this process cannot be efficiently parallelized, the difference between the time needed by a very powerful receiver and a standard receiver is limited. Hence, 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. 4.4 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.

Zero Knowledge Protocols As described in the introduction, in our constructions we do not rely on the generic zero knowledge protocols (for efficiency reasons). We will use them to describe how our protocols can be generalized (see Section 4.3). There is no space here to give a full introduction to zero-knowledge (the reader may consult, e.g., [27], for this). Let us only say that these protocols [28] are executed between a *prover* P and a *verifier* V. The goal of the prover is to convince the verifier that some instance x is a member of some language L (both P and V know (x, L)). The *zero-knowledge* property means that the verifier does not learn any information beyond the fact that $x \in L$ (even if he is malicious, i.e., he does not follow the rules of the protocol). Every protocol has also to be *sound* by which we mean that it is impossible to convince the

verifier that $x \in L$ if it is not the case (except with negligible probability). In our paper we actually need a stronger notion, namely the *zero-knowledge proofs of knowledge* [7]. 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., [27] 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. Section 4.3) 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, [22]), 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) [24]. 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)* [29, 20], 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 Appx. B.

We will use the additively-homomorphic public key encryption scheme introduced by Pascal Paillier [39]. Below, we describe only the properties of this scheme that are needed in this work. For more details the reader can consult, e.g, [39]. 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* [39]. 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 some value $x \in \{0,1\}^*$ the committer chooses some 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 (a commitment that can be open in two different ways would automatically form a collision for 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 [41]. 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 [41] 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. Since we do not use the non-standard transactions we will provide a simplified description that ignores this feature of Bitcoin. More on Bitcoin transactions can be found, e.g., in [15, 5]. 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*[3]. In the simplest case transaction T simply sends some amount $\text{฿}x$ (where x can be smaller than one[4]) from an address pk_0 (called an *input* of T) to an address pk_1 (called the *output* of T). The amount $\text{฿}x$ 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 $\text{฿}x$, 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 $[T] := (\texttt{TXid}(T'), \texttt{value} : \text{฿}x, \texttt{from} : pk_0, \texttt{to} : pk_1)$, 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 $([T], \mathsf{Sign}_{sk_0}([T]))$.

A transaction can also have multiple inputs and multiple outputs, but we do not describe this feature here, as we do not need it. Let us also say that the total value x of the transactions redeemed by T can be larger than the total value x' of the transactions that spend T. The difference $x - x'$ is called a *transaction fee*. On the other hand, obviously, it is not allowed to have $x' > x$. In this paper, for simplicity, we will describe our protocols with an assumption that $x' = x$, i.e., that the fees are equal to zero.

Another standard type of the transactions are the so-called *multisig* transactions. In this case $[T]$ has a form $(\texttt{TXid}(T'), \texttt{value} : \text{฿}x, \texttt{from} : pk_0, \texttt{to}$ "k-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 $1 \leq j \leq k$ it holds that $\mathsf{Vrfy}_{pk_{i_j}}([T''], \sigma_{i_j}) = \mathsf{ok}$. The multisig transactions are typically used to increase the security of the bitcoin storage: instead of storing one key, a user can generate n keys, and store each of them in a separate place. In this way even if $k - 1$ keys leak to the adversary, the money is still safe. On the other hand: if $n - k$ keys get lost the user can still spend the transaction, which provides some level of security in case of, e.g., the hardware on which the keys are stored is damaged or stolen. The "2-out-of-3" multisig transactions are also used with (partially) trusted third party that helps in mediating disputes.

3 On the malleability of the Bitcoin transactions and the ECDSA signatures

As described in Section 2, the Bitcoin transactions are identified by their hashes, more precisely: $\texttt{TXid}(T) = H(T)$ (where H is the SHA256 hash function). This can sometimes be a problem, since it turns out that often the adversary that knows T can compute T' that is *semantically equivalent* to T (i.e.: spends the same transaction, has the same value, and the same inputs and outputs), but is *syntactically different* from T. This is called *mauling* the transaction T. In a moment we will explain how mauling can be done, but let

[3] Technically: it is actually the hash of pk, but this is not relevant to this high-level description.

[4] The smallest currency unit in Bitcoin is $\text{฿}10^{-8}$.

us first say what are the consequences of this attack. The problem comes from the fact that in such a case $\texttt{TXid}(T) \neq \texttt{TXid}(T')$. Note that this does not imply that the adversary can steal money (as the output of T is the same as the output of T'). Still, it is undesirable in many situations. Firstly, it turns out [3] that several Bitcoin software clients are not prepared to handle situations when the transaction that appears on the ledger has a different identifier than the transaction that was originally posted. Secondly, and more importantly, it creates problems when creating Bitcoin contracts (see, e.g., [2, 3]).

There are actually several methods that allow to maul a transaction in Bitcoin. The simplest ones exploit the properties of the Bitcoin scripting language and we will not discuss them here. What is more relevant to this paper is the attack that is based on mauling the ECDSA signatures (the malleability of signatures was defined in Section 2). As it turns out these signatures are easily malleable: 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 the same public key). To see why this is the case observe that $\varphi : \mathbb{G} \to \mathbb{G}$ (where $\mathbb{G}$ denotes the elliptic curve) defined as $\varphi(h) = (-h)$ is an automorphism on $\mathbb{G}$ (which is easy to verify by checking the formulas for the group operation), and the reduction function f (cf. Section 2) is invariant with respect to taking this automorphism, i.e. it is such that

$$\forall_{h \in \mathbb{G}} f(h) = f(\varphi(h)) \tag{1}$$

(this is because f is defined as $f(x, y) = x$ and in $\mathbb{G}$ we have that $-(x, y) = (x, -y)$). Recall that the ECDSA verification algorithm takes as input a public key D, a signature (r, s), and a message z and checks whether

$$r = f(H(z) \cdot s^{-1} \cdot g + r \cdot s^{-1} \cdot D). \tag{2}$$

Now observe that if we run the check from Eq. (2) on $(r, -s)$ then the right hand side on (2) becomes

$$\begin{aligned} & f(-H(z) \cdot s^{-1} \cdot g - r \cdot s^{-1} \cdot D)) \\ = {} & f(\varphi(H(z) \cdot s^{-1} \cdot g + r \cdot s^{-1} \cdot D)) \\ = {} & f(H(z) \cdot s^{-1} \cdot g + r \cdot s^{-1} \cdot D) \\ = {} & r, \end{aligned}$$

so on input $(r, -s)$ the verification algorithm also returns ok. We now conjecture that this is the only way that the ECDSA signatures on the Secp256k1 curve can be mauled. One of the reasons why we believe that this assumption holds is that the attack described above works only since φ is an automorphism such that (1) holds. But, since Secp256k1 has j-invariant 0 (see [19]) we know (see [46], Section 2.7) that φ is its only non-trivial automorphism. Hence, no other attack of this type is possible.

It also seems that the Bitcoin community generally believes that there is no other method of mauling Bitcoin signatures. For example, the *Bitcoin Improvement Proposal (BIP) 0062* [47] lists 9 possible sources of malleability of Bitcoin transactions, among them the "Inherent ECDSA signature malleability", where the only method of mauling the signatures is the one described above (taking the negative of s). Also, the only countermeasure against the signature malleability mentioned in [47] is to disallow negating s. Technically this is done be requiring that $s \leq (p-1)/2$.

In order to make our signature scheme strongly-unforgeable we follow the guidelines from [47]. Namely, we assume that the only "legal" signatures have a form (r, s) such that $s \leq (p-1)/2$. Note that, since we want our protocols to work over the current version of Bitcoin, we cannot assume that the BIP 0062 was already implemented. We therefore do *not* assume it, and hence our protocols need to be ready to handle situations where $s > (p-1)/2$ (in case, e.g., one of the participants is malicious). 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:

(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. To do this, we would need to define a *family* of signature schemes parametrized by a security parameter. A natural example of such a family could be the ECDSA signatures defined over any prime field. We do not state this assumption formally, since anyway at the end we are interested only in the concrete case of the Secp256k1 curve.

We also note that our scheme remains secure even if some other methods of mauling the signatures become known, as long as they permit efficient "inversion" of the mauling function, i.e, after obtaining σ' which is a "mauled" version of σ it is possible to construct a polynomial-size set Σ such that $\sigma \in \Sigma$.

4 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 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 $f : \{0,1\}^* \to \{\mathsf{true}, \mathsf{false}\}$). 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. 4.3. First, however, we give an outline of our construction. The necessary ingredients are defined and constructed in Sections 4.1 and 4.2.

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[5] $[T_2]$ 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).

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

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 Sect. 4.3.

Secondly, the currently-known protocols for distributed signing with the ECDSA signatures are rather complicated and involve costly generic zero-knowledge techniques [34] (see also [26]). 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, [32]). 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 Sections 4.1 and 4.2. 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), 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. 4.3. The outline of the $\mathsf{SellWitness}_f$ protocol and the subprotocols is presented on Fig. 1.

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 the GenMsg_T procedure to produce transactions T_1 and T_2.
3. The parties run the USG protocol to sign T_2 using all a shared keys (only the Seller learns the signatures) 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 if the Seller does not broadcast T_2).

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

4.1 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 $\mathsf{SharedKGen}$, 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 presented on Figure 2 and it uses a commitment scheme $\mathsf{COM} = (\mathsf{Commit}, \mathsf{Open})$. The protocol is very similar to the classic actively-secure key generation protocols for the discrete log signatures [40]. It is secure against active adversaries. Since it is a standard argument we do not prove its security here.

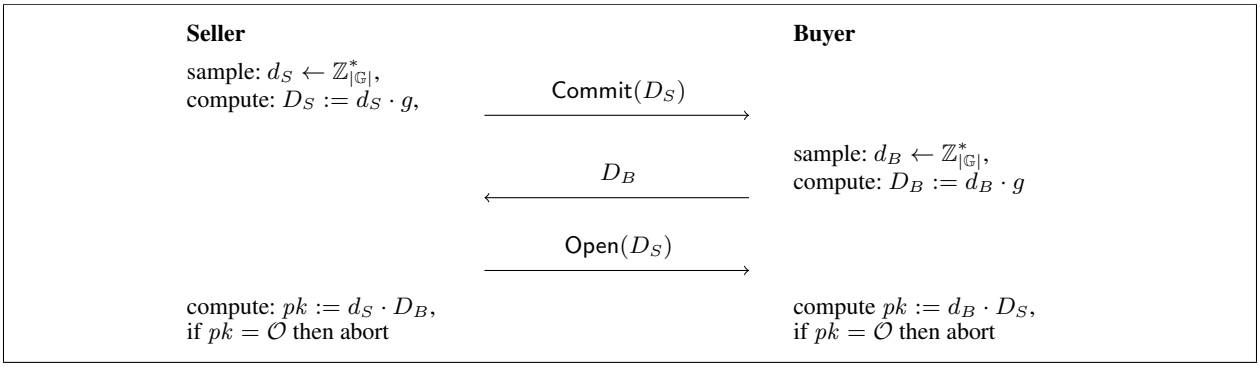

Fig. 2: The $\mathsf{SharedKGen}(1^\lambda)$ protocol.

4.2 The Unique Signature Generation Protocol

After the parties generate a key pairs $(sk^1, pk^1), \ldots, (sk^a, pk^a)$ using the $\mathsf{SharedKGen}$ 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 $\mathsf{SharedKGen}$ 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 $(\hat{d}^i_S, \hat{d}^i_B)$ 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 $\hat{d}^1_S, \ldots, \hat{d}^b_S$ 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 $\hat{d}^i_S$ (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. 3. We assume that before it is executed the parties run the $\mathsf{SharedKGen}$ procedure (cf. Fig. 2) a times (on input 1^λ). We denote these executions as $\mathsf{SharedKGen}^i(1^\lambda)$ for $i = 1, \ldots, a$. 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_S^i, d_B^i) be the respective shares).

The USG protocol uses as a subroutine the protocol $\mathsf{KSignGen}$ from Fig. 4. 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 Appendix C.

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. 3 is* $(\epsilon, \hat{b})$*-secure for* $\epsilon = (b/a)^{\hat{b}}$.

4.3 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. 4). 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 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 $\mathsf{VerifyResponse}_\mathcal{F}(B_\mathcal{F}, c_\mathcal{F}, r_\mathcal{F}) \in \{\mathsf{true}, \mathsf{false}\}$.

1. The Buyer chooses a random subset $\mathcal{J} \subset \{1, \ldots, a\}$, such that $|\mathcal{J}| = a - b$. Let $\{j_1, \ldots, j_b\}$ denote the set $\{1, \ldots, a\} \setminus \mathcal{J}$.
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. 4. As a result of each such execution, the Seller is committed to $S^i = F(\sigma^i)$ and timed-committed to d_S^i.
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_S^j, and sends σ^j, k_S^j and sk_{AH}^j 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 $j \in \mathcal{J}$):
   - $\mathsf{Vrfy}_{pk^j}(z, \sigma^j) = \mathsf{ok}$,
   - $F(\sigma^j) = S^j$,
   - $d_S^j \cdot d_B^j \cdot g = pk^j$, and
   - $\mathsf{Dec}_{sk_{\mathrm{AH}}^j}(c_S^j) = d_S^j$.
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. 3: The USG protocol. Note that it uses the $\mathsf{KSignGen}(1^\lambda)$ procedure, depicted on Fig. 4, as a subroutine. Recall also that we assume that before the USG protocol is executed, the parties run the $\mathsf{SharedKGen}$ procedure (cf. Fig. 2) a times (on input 1^λ). As a result of each such execution, both parties learn the public keys $pk^1, \ldots, pk^a$ and they secret-share the corresponding secret keys $sk^1, \ldots, sk^a$ (let $(d_S^1, d_B^1), \ldots, (d_S^a, d_B^a)$ be the respective shares).

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. $\mathsf{Extract}_{\mathcal{F}}(B_{\mathcal{F}}, c_{\mathcal{F}}^1, r_{\mathcal{F}}^1, c_{\mathcal{F}}^2, r_{\mathcal{F}}^2) = x'$ and $f(x') = \mathsf{true}$ if only $\mathsf{VerifyResponse}_{\mathcal{F}}(B_{\mathcal{F}}, c_{\mathcal{F}}^i, r_{\mathcal{F}}^i) = \mathsf{true}$ for $i = 1, 2$ and $c_{\mathcal{F}}^1 \neq c_{\mathcal{F}}^2$. 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-`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 $k \neq j_1, \ldots, j_\lambda$.

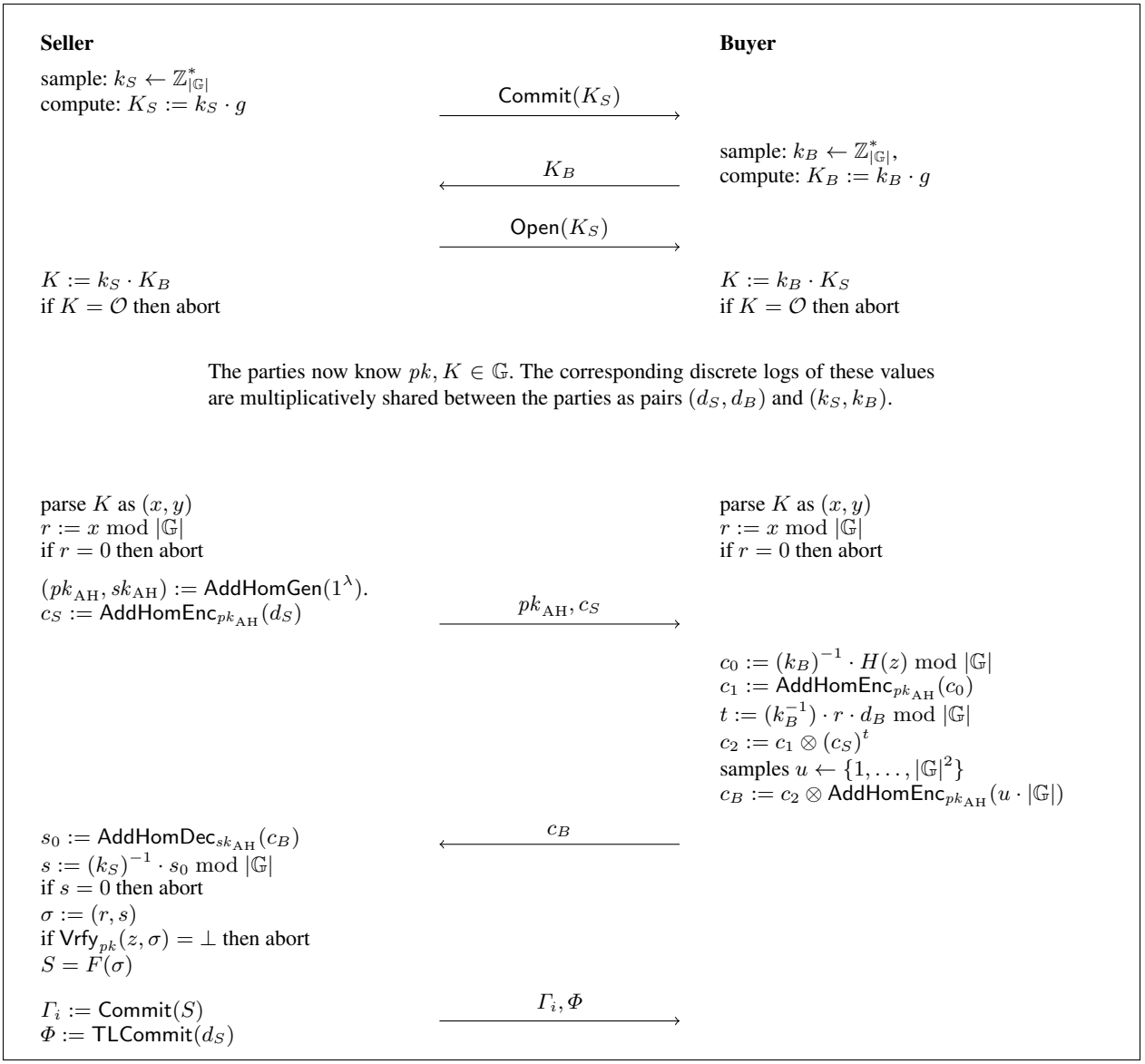

Fig. 4: The $\mathsf{KSignGen}(1^\lambda)$ procedure. Recall that $\mathbb{G}$ is an elliptic curve group for ECDSA, and $(\mathsf{AddHomGen}, \mathsf{AddHomEnc}, \mathsf{AddHomDec})$ is a Paillier encryption scheme which is additively homomorphic over $\mathbb{Z}_n$, where $n > 2 \cdot |\mathbb{G}|^4$.

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. 5. We have the following lemma, its proof appears in Appendix D.

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. $f(x) =$ true of the required form. Then the* $\mathsf{SellWitness}_f$ *constructed on Fig. 5 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 $\mathcal{J}^i \subset \{1, \ldots, 2\lambda\}$ 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 $\hat{\sigma}_1, \ldots, \hat{\sigma}_b$ 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. 5: The $\mathsf{SellWitness}_f$ protocol.

Construction using the OP_CHECKLOCKTIMEVERIFY operator. Recently a new instruction has been added to the Bitcoin scritping language — OP_CHECKLOCKTIMEVERIFY. Earlier dealing with time constraints in Bitcoin scripts was cumbersome and problematic. Now, using the OP_CHECKLOCKTIMEVERIFY operator one can easily create a transaction that can be redeemed by the receiver only after some specified time. It can also be used in more complicated scripts as a fuse for a transaction funder — if the other parties leave the protocol then the funder of a transaction can get his money back after some predefined time.

If we assume that the transactions using the OP_CHECKLOCKTIMEVERIFY operator are standard in a given cryptocurrency then we can simplify our protocol. What we exactly need is that the following type of transactions is standard: T can be redeemed by a "k-out-of-n" multisig $\hat{pk}_1, \ldots, \hat{pk}_n$ or after τ seconds by pk. In that case we can use this script in a transaction T_1 s.t. $k = n = b$ and $pk = pk_B$. Now the Buyer will be able to take his funds after τ seconds, so we can remove the timed commitments entirely from the USG and the $\mathsf{SellWitness}_f$ protocols.

4.4 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. 6 — a zero knowledge proof of knowledge of the factorization of the RSA modulus. We now have the following lemma, whose proof appears in Appendix E.

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. 6 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

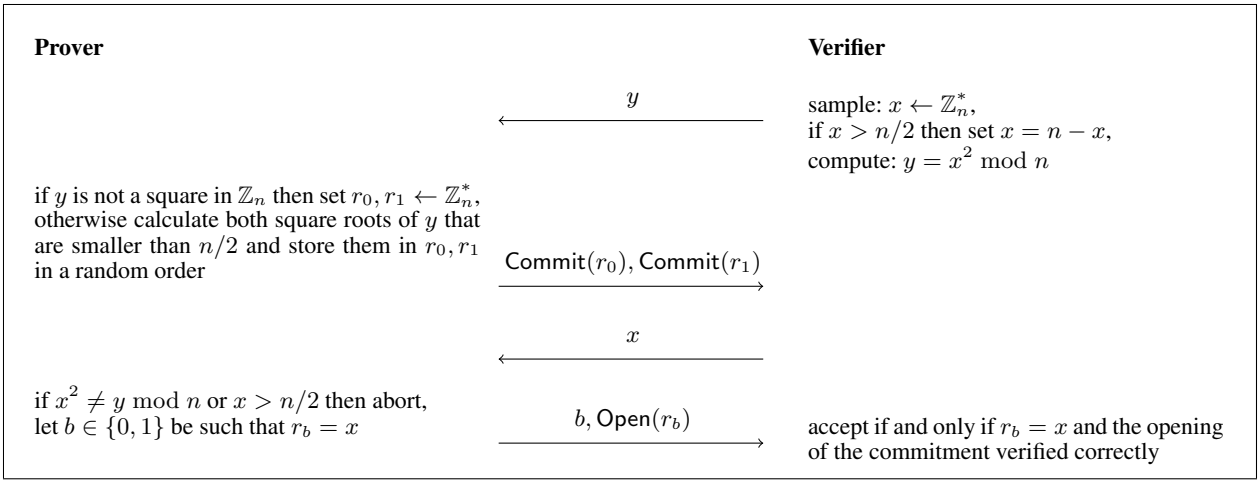

Fig. 6: The $\mathsf{ZKFactorization}(n)$ protocol

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 `http://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 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 minute — the running time of the USG protocol is about 33 seconds and Step 2 in the $\mathsf{SellWitness}_f$ protocol takes about 28 seconds. The numbers are an average over 10 runs of the algorithm using a single thread on a standard personal computer. As this is not a protocol that users may want to run frequently, we believe that this running time is satisfactory. We note that the running time could be improved by using multiple threads. Additional measurements are presented on Figures 7 and 8 in Appx. G.

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[6]. Taking into account time needed to post messages on the blockchain the running our protocol takes on average 2 hours, we have to have at least $\tau_0 = 5$ hours, so τ_1

[6] It takes on average 10 minutes for a transaction to be included into the blockchain but the users are advised to wait for 6 blocks ($\approx$ 1 hour) on top of the transaction to have more assurance that it will not be withdrawn.

should be set to 50 hours. 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 $t = 2^{37}$. The measurements of the time of the squaring algorithm are presented on Fig. 9 in Appx. G.

4.5 Protocol for trading across chains

As mentioned at the beginning of this paper, one of the interesting examples of the smart contracts is Trading across chains. Imagine a situation when Alice has her funds in one cryptocurrency $\mathfrak{A}$, e.g. she has b bitcoins, and Bob has his funds in another cryptocurrency $\mathfrak{B}$, e.g. he has z zerocoins. The problem is to exchange they funds.

There is a well known Bitcoin protocol (a contract) that solves this problem using advanced Bitcoin scripting language. However, almost all of the existing cryptocurrencies, including Zerocash, do not have advanced scripting language as Bitcoin so Alice and Bob cannot use that protocol in real life. Now we will sketch a protocol that can be used for trading across chains and is using only standard transactions. It is very similar to the protocol from the previous sections and it is also using only Cut and Choose technique.

Sketch of the protocol We describe this protocol assuming that in both cryptocurrencies $\mathfrak{A}$ and $\mathfrak{B}$ the "1-out-of-b" and "b-out-of-$(2b-1)$" multisig transactions are standard. If in any of them only single signature transactions are possible (the "1-out-of-1" transactions) then we will use $b = 1$ in the protocol.

At the beginning Bob samples his secret shares d_B^i and calculates $D_B^i = d_B^i \cdot g$ for $1 \leq i \leq a$ (using the group $(\mathbb{G}, \mathcal{O}, g, +)$ from the cryptocurrency $\mathfrak{A}$) and he commits to each D_B^i. Also Alice similarly samples her secret shares d_A^i, calculates $D_A^i = d_A^i \cdot g$ for $1 \leq i \leq a$ and sends each D_A^i to Bob. These values will be used later in creating the public keys for Alice in the cryptocurrency $\mathfrak{A}$.

Then the parties execute the USG protocol (Alice plays the role of the Seller and Bob plays the role of the Buyer) using the cryptocurrency $\mathfrak{B}$ and procedure GenMsg_T the same as in the $\mathsf{SellWitness}$ protocol. The only difference is that at the end of the $\mathsf{KSignGen}$ procedure Alice additionally sends $x^i = d_A^i \oplus F(\sigma^i)$ to Bob and Bob in the step 6 of the USG protocol verifies that she did it honestly.

So now Bob knows transaction T_B^1 that sends his funds to a "b-out-of-$(2b-1)$" multisig using b public keys generated and $(b-1)$ of his own public keys and both parties know T_B^2 that redeems T_B^1 and sends the funds to Alice. Alice knows signatures σ^i and she will be able to use them to broadcast T_B^2 and after that Bob will learn values d_A^i.

Now Bob opens his commitments to values D_B^i and calculates timed commitments to each d_B^i (with much longer force-opening time than timed commitments from Alice) and sends all the time-locked puzzles to Alice. Alice verifies the commitments and chooses a random set $\mathcal{J}_A \subset \{1, \ldots, a\}$ of size $a - b$ and sends it to Bob. Bob opens all the timed commitments for $j \in \mathcal{J}_A$ and Alice verifies them.

The parties set public keys for Alice as $pk_A^j = d_A^{k_j} \cdot D_B^{l_j} = d_B^{l_j} \cdot D_A^{k_j}$ where $k_1, \ldots, k_b$ are indices not present in $\mathcal{J}_A$ and $l_1, \ldots, l_b$ are indices not present in the set $\mathcal{J}_B$ chosen by Bob in the USG protocol. Now Alice creates and broadcasts T_A^1 that spends her funds and sends them to a multisig "1-out-of-b" using public keys $pk_A^1, \ldots, pk_A^b$. After T_A^1 is included into the blockchain Bob broadcasts (assuming T_A^1 is correct) transaction T_B^1.

After T_B^1 is included into the blockchain Alice can use T_B^2 and all the signatures σ^i to claim Bob's funds. Then Bob can calculate any of the d_A^i using σ^i and x^i to get Alice's funds. And if Alice do not claim Bob's funds then Bob after solving one of his time-locked puzzles can get his funds back[7]. And after some time Alice can solve one of her time-locked puzzles to get her funds back[8].

Sketch of the security of the protocol appears in the Appendix E.

[7] Actually as some of the time-locked puzzles may be forged, Bob should open all of his puzzles in parallel.

[8] Remember that her time-locked puzzles require much more time to be solved than Bob's, so Bob will be able to open all his puzzles before Alice opens at least one of hers.

References

[1] J. H. An, Y. Dodis, and T. Rabin. "On the Security of Joint Signature and Encryption". In: *EUROCRYPT 2002*. 2002.

[2] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. "Fair Two-Party Computations via Bitcoin Deposits". In: *BITCOIN*. 2014.

[3] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. "On the Malleability of Bitcoin Transactions". In: *BITCOIN*. 2015.

[4] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. "Secure Multiparty Computations on Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. 2014.

[5] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. 1st. 2014.

[6] A. Back and I. Bentov. "Note on fair coin toss via Bitcoin". In: *CoRR* (2014).

[7] M. Bellare and O. Goldreich. "On Defining Proofs of Knowledge". In: *CRYPTO'92*. 1993.

[8] M. Bellare and P. Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *ACM CCS 93*. 1993.

[9] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. 2014.

[10] I. Bentov and R. Kumaresan. "How to Use Bitcoin to Design Fair Protocols". In: *CRYPTO 2014, Part II*. 2014.

[11] *Bitcoin Wiki: Block size limit controversy*.

[12] *Bitcoin Wiki: Constract*.

[13] *Bitcoin Wiki: Script*.

[14] *Bitcoin Wiki: Secp256k1*.

[15] *Bitcoin Wiki: Transaction*.

[16] *Bitcoin Wiki: Zero Knowledge Contingent Payment*.

[17] D. Boneh and M. Naor. "Timed Commitments". In: *CRYPTO 2000*. 2000.

[18] D. Boneh, E. Shen, and B. Waters. "Strongly Unforgeable Signatures Based on Computational Diffie-Hellman". In: *PKC 2006*. 2006.

[19] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow. "Elliptic Curve Cryptography in Practice". In: *FC 2014*. 2014.

[20] D. R. L. Brown. *Standards for Efficient Cryptography SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0*. 2010.

[21] D. Brown. "The Exact Security of ECDSA, January 2001". In: *IEEE* (1998).

[22] I. Damgard. *On Sigma-Protocols*. a manuscript available at `http://www.cs.au.dk/~ivan/Sigma.pdf`. 2015.

[23] C. Dwork and M. Naor. "Pricing via Processing or Combatting Junk Mail". In: *CRYPTO'92*. 1993.

[24] A. Fiat and A. Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *CRYPTO'86*. Aug. 1987.

[25] P. D. Filippi. *Tomorrow's apps will come from brilliant (and risky) Bitcoin code*. Wired magazine.

[26] S. Goldfeder, R. Gennario, H. Kalodner, J. Bonneau, E. W. Felten, J. A. Kroll, and A. Narayanan. *Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme*. manuscript. 2015.

[27] O. Goldreich. *Foundations of Cryptography: Volume 1*. 2006.

[28] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof Systems". In: *SIAM J. Comput.* 1 (Feb. 1989). ISSN: 0097-5397.

[29] D. Johnson, A. Menezes, and S. Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International Journal of Information Security* 1 (2001).

[30] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. 2007.

[31] R. Kumaresan, T. Moran, and I. Bentov. "How to Use Bitcoin to Play Decentralized Poker". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 2015.

[32] Y. Lindell and B. Pinkas. "Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer". In: *TCC 2011*. 2011.

[33] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. "Demystifying Incentives in the Consensus Computer". In: *ACM CCS 15*. 2015.

[34] P. MacKenzie and M. K. Reiter. "Two-party generation of DSA signatures". English. In: *International Journal of Information Security* 3-4 (2004). ISSN: 1615-5262.

[35] G. Maxwell. *The first successful Zero-Knowledge Contingent Payment*. 2016.

[36] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. `http://bitcoin.org/bitcoin.pdf`. 2009.

[37] H. Pagnia and F. C. Gartner. *On the impossibility of fair exchange without a trusted third party*. Tech. rep. Darmstadt University of Technology, 1999.

[38] H. Pagnia, H. Vogt, and F. C. Gärtner. "Fair Exchange". In: *Comput. J.* 1 (2003).

[39] P. Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *EUROCRYPT'99*. 1999.

[40] T. P. Pedersen. "A Threshold Cryptosystem without a Trusted Party (Extended Abstract) (Rump Session)". In: *EUROCRYPT'91*. 1991.
[41] R. L. Rivest, A. Shamir, and D. A. Wagner. *Time-lock Puzzles and Timed-release Crypto*. Tech. rep. 1996.
[42] D. Shanks. "Five number-theoretic algorithms". In: *Proceedings of the Second Manitoba Conference on Numerical Mathematics (Univ. Manitoba, Winnipeg, Man., 1972)*. 1973.
[43] J. Stern, D. Pointcheval, J. Malone-Lee, and N. P. Smart. "Flaws in Applying Proof Methodologies to Signature Schemes". In: *CRYPTO 2002*. 2002.
[44] N. Szabo. "Formalizing and Securing Relationships on Public Networks". In: *First Monday* 9 (1997). ISSN: 13960466.
[45] P. Todd. *OP_CHECKLOCKTIMEVERIFY*. Bitcoin Improvement Proposal 0062, available at `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`.
[46] L. C. Washington. *Elliptic Curves: Number Theory and Cryptography, Second Edition*. 2nd ed. 2008.
[47] P. Wuille. *Bitcoin Improvement Proposal 062: Dealing with malleability*.

A Definitions omitted in the main body

A function $\mu : \mathbb{N} \to \mathbb{R}$ is *negligible (in x)* if for every positive integer c there exists an integer N_c such that for all $x > N_c$ we have that $|\mu(x)| < \frac{1}{x^c}$. We say that a sequence $\{\mathcal{E}_\lambda\}_{\lambda=1}^{\infty}$ of events *happens with overwhelming probability* if $1 - \mathbb{P}(\mathcal{E}_\lambda)$ is negligible (as a function of λ).

A *public-key encryption scheme* is a tuple $(\mathsf{EncGen}, \mathsf{Enc}, \mathsf{Dec})$ of algorithms such that the *key generation algorithm* (EncGen) takes as input a security parameter 1^λ, and produces as output a key pair $(pk, sk) \in (\{0,1\}^*)^2$. A pair (pk, sk) that is an output of EncGen (for some 1^λ and the internal randomness of EncGen) will be called a *valid key pair*. The *encryption algorithm* Enc takes as input pk, a message $z \in \{0,1\}^*$ and produces as output a ciphertext $c = \mathsf{Enc}_{pk}(z) \in \{0,1\}^*$. The *decryption algorithm* Dec takes as input sk, c, and produces as output $z' = \mathsf{Dec}_{sk}(c)$. We require that it is always the case that $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(z)) = z$ (for every valid (pk, sk)). We may also restrict the domain of Enc, and allow that in some cases the algorithm $\mathsf{Enc}_{pk}(z)$ produces as output $\perp$. If this is the case then we do not require that $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(z)) = z$, and we say that z is an *invalid input for pk*. Otherwise (i.e. when $\mathsf{Enc}_{pk}(z) \neq \perp$) we sat that z is a *valid input for pk*.

To define security of an encryption scheme $(\mathsf{EncGen}, \mathsf{Enc}, \mathsf{Dec})$ consider a polynomial time adversary $\mathcal{A}$ that receives as input (pk, sk) (where $(pk, sk) := \mathsf{EncGen}(1^\lambda)$), and produces a pair of messages (m_0, m_1). He then receives $\mathsf{Enc}_{pk}(m_b)$ for a random $b \leftarrow \{0,1\}$, and produces $b' \in \{0,1\}$. We say that $\mathcal{A}$ *won* if $b' = b$. We say that $(\mathsf{EncGen}, \mathsf{Enc}, \mathsf{Dec})$ is *semantically secure* if for every polynomial time $\mathcal{A}$ the probability that he wins is at most $1/2 + \mu(\lambda)$, where μ is some negligible function (in other words: $\mathsf{Enc}_{pk}(z_0)$ and $\mathsf{Enc}_{pk}(z_1)$ are *computationally indistinguishable*).

The definition of a *symmetric-key encryption scheme* is similar, except that there is only one key k, and it is usually sampled uniformly at random from some space $\mathcal{K}$ (that depends on 1^λ). The adversary does not learn k, but can get some partial information about it by adaptively choosing messages $m_1, \ldots, m_n$ and learning $\mathsf{Enc}_k(m_1), \ldots, \mathsf{Enc}_k(m_n)$.

A *commitment scheme* is a pair of algorithms $(\mathsf{Commit}, \mathsf{Open})$. The *commitment algorithm* Commit takes as input a message $z \in \{0,1\}^*$, and produces as output $c = \mathsf{Commit}(z)$. Let d_c be the randomness used during the generation of the commitment c. The *opening algorithm* Open takes as input (c, z, d) and outputs $\mathsf{Open}(c, z, d) \in \{0,1\}^* \cup \{\perp\}$. We require that always $\mathsf{Open}(\mathsf{Commit}(z), z, d_c) = z$.

A commitment scheme will usually be executed between two parties: a *committer* and a *receiver*. In the *commitment phase* the committer computes $c := \mathsf{Commit}(z)$ and sends it to the receiver. Later, in the *opening phase* the committer sends (z', d') to the receiver, and the receiver outputs $z'' = \mathsf{Open}(c, z', d')$. A commitment scheme is *secure* if it is *binding* and *hiding*. The "hiding" property means that the receiver has no information about z before the opening phase started. More precisely: for every $z_0, z_1 \in \{0,1\}^*$ we have that $\mathsf{Commit}(z_0)$ and $\mathsf{Commit}(z_1)$ are computationally indistinguishable. The "binding" property means that, no matter how a malicious (but polynomial-time bounded) committer behaves, after the commitment phase there exists at most one value $z' \in \{0,1\}^*$ such that the receiver outputs z' (or outputs $\perp$).

B The ECDSA algorithm

For a security parameter 1^λ the ECDSA key generation algorithm $\mathsf{SignGen}$ chooses an elliptic curve group $(\mathbb{G}, \mathcal{O}, g, +)$ over a prime field Z_p, where $\mathcal{O}$ is the neutral element, g is the generator of $\mathbb{G}$, and the order $|\mathbb{G}|$ of $\mathbb{G}$ is a prime number such that $\lceil \log_2 |\mathbb{G}| \rceil = \lambda$ (the details on how this choice is done are not relevant to this paper). Then, $\mathsf{SignGen}$ samples a random $d \leftarrow \mathbb{Z}_{|\mathbb{G}|}$, and computes $D := d \cdot g$. The generated secret key is $(d, (\mathbb{G}, \mathcal{O}, g, +))$, and the public key is $(D, (\mathbb{G}, \mathcal{O}, g, +))$.

Let H be a hash function, and let $f : \mathbb{G} \to \mathbb{Z}_{|\mathbb{G}|}$ be a *reduction function* that we will define in a moment. In order to sign a message $z \in \{0,1\}^*$ with key $(d, (\mathbb{G}, \mathcal{O}, g, +))$ the signing algorithm first chooses a random $k \leftarrow \mathbb{Z}_{|\mathbb{G}|}$, and then computes $r = f(k \cdot g)$ and $s = k^{-1}(H(z) + d \cdot r) \bmod |\mathbb{G}|$. If $r = 0$ or $s = 0$ then the algorithm aborts (and needs to be restarted with fresh randomness). Otherwise, the signing algorithm outputs (r, s). The verification algorithm on public key $(D, (\mathbb{G}, \mathcal{O}, g, +))$ and input z and (r, s) first checks if r and s are non zero elements of $\mathbb{Z}_{|\mathbb{G}|}$ and then verifies if $r = f(H(z) \cdot s^{-1} \cdot g + r \cdot s^{-1} \cdot D)$ (where the operations in the last formula are in $\mathbb{G}$)[9]. If this holds, then the verification algorithms outputs ok, and otherwise it outputs $\perp$. It is straightforward to verify that a verification of a correctly signed message will always yield ok.

What remains is to construct the reduction function $f : \mathbb{G} \to \mathbb{Z}_{|\mathbb{G}|}$. In ECDSA this is defined by exploiting the structure of the elliptic curve group constructed over prime fields. Recall that every element of $\mathbb{G}$ has a form $(x, y) \in Z_p^2$. The function f on input (x, y) ignores y and produces as output $f(x, y) = x \bmod |\mathbb{G}|$. The ECDSA algorithm with f defined in this way is widely believed to be secure for several choices of the elliptic curves. It has also been proven existentially unforgeable under a chosen message attack, provided some assumptions hold (see, e.g, [21, 43]). The malleability of the ECDSA scheme is a more subtle issue. We address it in Section 3.

C Proof of Lemma 1

Assume first that the Buyer is honest and that the protocol has not been aborted. Let us call an execution i of the $\mathsf{KSignGen}^i(1^\lambda)$ *invalid*, if at least one of the following happened:

(a) $\mathsf{Dec}_{sk^i_{\mathrm{AH}}}(c^i_S) \neq d^i_S$,
(b) the solution of the Φ_i is not equal to d^i_S i.e. $\mathsf{TLForceOpen}(\Phi_i) \cdot d^i_B \cdot g \neq pk^i$,
(c) the Seller can open Γ_i into a value different than $F(\hat{\sigma}_i)$ for a valid signature $\hat{\sigma}_i$ on z.

Call $\mathsf{KSignGen}^i(1^\lambda)$ *uncovered* if $i \in \mathcal{J}$. If an execution i is invalid and $i \in \mathcal{J}$ then the Buyer will abort the protocol with probability $1 - \mu_0(\lambda)$ where μ_0 is a negligible function. That is because we assume that the underlying cryptographic primitives are secure, so if the Seller can open a commitment (or timed commitment) to a certain value, then she cannot open it to a different value. So as she opens Φ_i or Γ_i (if she does not open it then the Buyer will abort) in case respectively (b) or (c) then the Buyer will find out that she cheated and he will abort. And in case (a) there is only one secret key corresponding to pk^i_{AH} so the Seller has to reveal it and now the Buyer will also find out that the Seller cheated.

The Seller and the Buyer execute the $\mathsf{KSignGen}^i(1^\lambda)$ procedure a times and the Seller reveals $(a - b)$ random executions (chosen by the honest Buyer). We assume that the Buyer did not abort the protocol, so none of the uncovered executions was invalid (except with negligible probability). So the probability that at least $\hat{b}$ executions were invalid, but the Buyer did not abort the protocol, is at most (ignoring the negligible

[9] Recall that $\mathbb{G}$ is an additive group, and hence $a \cdot g$ (for $a \in \mathbb{Z}_p$ and $g \in \mathbb{G}$) denotes $\overbrace{g + \cdots + g}^{a \text{ times}}$. Also $a \cdot b^{-1} \cdot g$ (for $a, b \in Z_p$ and $g \in \mathbb{G}$) denotes $c \cdot g$, where $c = a \cdot b^{-1} \bmod p$.

probabilities):

$$\binom{a-\hat{b}}{a-b}\Big/\binom{a}{a-b}=\prod_{j=0}^{\hat{b}-1}\frac{b-i}{a-i}\leq\left(\frac{b}{a}\right)^{\hat{b}}.$$

We now show that if an execution i is not invalid then the probability that the Seller breaks key i is negligible. We will prove that by showing that the Seller does not learn anything about the secret key sk^i except of the corresponding public key pk^i and a single signature σ_i on a message z. It will be sufficient because we assume that the ECDSA scheme used in the construction of the USG is a non-malleable signature scheme — in a not invalid execution Γ_i is a commitment of $F(\sigma_i)$, where σ_i is a signature on z and this is the only signature that the Seller knows.

In the first phase of the $\mathsf{KSignGen}^i(1^\lambda)$ the Seller chooses value d_S^i independently to d_B^i so (d_S^i, d_B^i) is a multiplicative secret sharing of the secret key sk^i (she cannot set $d_S^i=0$, in that case the Buyer would abort). By the same argument (k_S^i, k_B^i) is a multiplicative secret sharing of the secret random value $k^i=k_S^i\cdot k_B^i$. In this phase the Seller learns D_B^i and K_B^i but these values can be computed by her given the public key pk^i and the signature $\sigma^i=(r^i,s^i)$ by calculating $D_B^i=(d_S^i)^{-1}\cdot pk^i$ and $K_B^i=(d_S^i)^{-1}\cdot K^i$ and K^i can be easily computed from the signature and the public key. So she has no information on neither sk^i nor K^i but the public key pk^i and the signature σ^i — values D_B^i and K_B^i do not give her any additional information.

The only additional information that the Seller learns during the $\mathsf{KSignGen}^i(1^\lambda)$ procedure is the value s_0^i. Observe first that $s_0^i \bmod |\mathbb{G}|$ can be computed by the Seller from the signature: $s_0^i \bmod |\mathbb{G}| = k_S^i \cdot s^i \bmod |\mathbb{G}|$. However in theory s_2^i can give the Seller more information than $s_0^i \bmod |\mathbb{G}|$. Now we will show why this is not the case.

We assumed that the execution i is not invalid, so $\mathsf{Dec}_{sk_{\mathrm{AH}}^i}(c_S^i)=d_S^i$ is smaller (as an integer) than $|\mathbb{G}|$. It follows that $\mathsf{Dec}_{sk_{\mathrm{AH}}^i}(c_2^i)$ is smaller than $2|\mathbb{G}|^2$. The Buyer added to this value random u^i times $|\mathbb{G}|^2$, so

$$\begin{aligned}\mathbb{P}(s_0^i<2|\mathbb{G}|^2\vee s_0^i>|\mathbb{G}|^3)<\\ <\mathbb{P}(u<2|\mathbb{G}|)+\mathbb{P}(u>|\mathbb{G}|^2-2|\mathbb{G}|)=\\ =O\left(|\mathbb{G}|^{-1}\right)=\mathrm{negligible}(\lambda),\end{aligned}$$

and for any $2|\mathbb{G}|^2\leq x\leq|\mathbb{G}|^3$ we have

$$\mathbb{P}(s_0^i=x)=\mathbb{P}(u=x-\mathsf{Dec}_{sk_{\mathrm{AH}}^i}(c_2^i))=|\mathbb{G}|^{-2}.$$

That proves that with overwhelming probability s_0^i is just $s_0^i \bmod |\mathbb{G}|$ plus $|\mathbb{G}|$ times something uniformly random, so it gives the Seller no more information.

Summarizing, and ignoring the negligible probabilities, with probability at most $(b/a)^{\hat{b}}$ the Seller can create undetected $\hat{b}$ invalid executions, in each of them she breaks the respective key i. The rest of the executions are not invalid and in them the Seller does not break the corresponding keys.

We can now assume that the Seller is honest and that the protocol has not been aborted. As it was in case of dishonest the Seller after the first phase the only information about the sk^i known by the Buyer is the public key pk^i. That is so because D^i can be computed from the public key and for the Buyer K^i is just a random point in $\mathbb{G}$. Later the Buyer learns pk_{AH}^i that is randomly generated and c_S^i that by the security of the Paillier encryption scheme he cannot distinguish from the encryption of a random value, so it gives him nothing. At the end he learns the commitment to the $S^i=F(\sigma^i)$, for a hash function F modeled as a random oracle so by the hiding property of the commitment scheme he known nothing about the S^i and similarly by the properties of the timed commitment the Buyer will open the commitment to d_S^i and thus learn the secret key sk^i only after time τ (both with probability 1 minus negligible).

This concludes the proof that this USG is $(\epsilon,\hat{b})$-secure for $\epsilon=\left(\frac{b}{a}\right)^{\hat{b}}$. □

D Proof of Lemma 2

Assume first that the Buyer is honest and the Seller is not. The only way how the Buyer can lose his funds is by the transaction T_1, as only he knows the secret keys required to spend his money. T_1 can be either redeemed by the Seller or it can just lock the Buyer's funds forever. We know from lemma 1 that with probability at most $\left(\frac{b}{a}\right)^b + \mu_0(\lambda)$ for a negligible μ_0 the Seller can cheat in all b chosen executions of the $\mathsf{KSignGen}$ procedure and not be caught.

So with probability at least $1 - \left(\frac{b}{a}\right)^b - \mu_0(\lambda)$ at least one of the b chosen executions of the $\mathsf{KSignGen}$ procedure was completed honestly by the Seller. Let the index of this execution be i. That means that Φ^i is an honest timed commitment of d_S^i so after force-opening this puzzle the Buyer will learn the secret key $\hat{sk}^i$. But he can easily compute $b-1$ correct signatures on a transaction redeeming T_1, because he owns $b-1$ out of $2b-1$ public key from the multisig escrow of the transaction T_1. So with additional signature created using $\hat{sk}^i$ he will have b signatures so he will be able to redeem transaction T_1 and get back his fund (in case when e.g. the Seller does not redeem T_1 by herself). So his funds cannot be locked forever (except with probability $\left(\frac{b}{a}\right)^b + \mu_0(\lambda)$), he will take them back after spending τ_1 seconds on force-opening Φ_i.

Note also that the same is true when transaction T_1 is mauled before it is included into the blockchain. In that case the Seller will not be able to use signatures $\hat{\sigma}^i$ but the Buyer will be able to get his funds back. This is the only problem that the malleability of transactions can create and it does not affect the security of the protocol.

Assume now that the Seller indeed redeems transaction T_1. The only way how she can do it is by using T_2 and all the signatures $\hat{\sigma}^1, \ldots, \hat{\sigma}^b$. So in that case the Buyer learns σ^i. As the execution number i of the $\mathsf{KSignGen}$ protocol was honest, then the Buyer will be able to compute the secrets $S^{i,j}$ for $j = 1, \ldots, 2\lambda$. Now we will compute what is the probability, that the Buyer will not be able to extract the witness x' although he did not abort in the second Cut and Choose verification in the step 2 in the $\mathsf{SellWitness}_f$ protocol.

Lets call $j \in \{1, \ldots, 2\lambda\}$ valid if $\mathsf{VerifyResponse}(B^{i,j}, c_k^{i,j}, r_k^{i,j}) = \mathsf{true}$ for both $k = 1$ and $k = 2$ and $c_0^{i,j} \neq c_1^{i,j}$, where $(c_k^{i,j}, r_k^{i,j}) := \mathsf{Dec}_{S^{i,j}}(\gamma_k^{i,j})$. Otherwise we will call j invalid. Assume first that more then λ indices j are valid. That means that at least one index $j \notin \mathcal{J}^i$ is valid. In that case the Buyer will learn $S^{i,j}$ because execution number i was honest so he will learn $c_k^{i,j}$ and $r_k^{i,j}$ for $k = 1, 2$ and he will be able to compute $x' = \mathsf{Extract}_{\mathcal{F}}(B^{i,j}, c_1^{i,j}, r_1^{i,j}, c_2^{i,j}, r_2^{i,j})$ and from the definition of the extractor function $f(x') = \mathsf{true}$.

In the second case we have at least λ invalid indices. If j is invalid and $j \in \mathcal{J}^i$ then with probability at least $\frac{1}{2}$ the Buyer will abort for the pair (i, j). That is true, because the Buyer's challenge $c_B^{i,j}$ is independent of $\gamma_k^{i,j}$, so even if in one of the $\gamma_k^{i,j}$ there is a correct (challenge, response) pair then it will be different than $c_B^{i,j}$ with probability $\frac{1}{2}$. Let X be the event that the Buyer aborts and $Y(r)$ an event when at least r invalid indices are present in $\mathcal{J}^i$. Then

$$\mathbb{P}(X) \geq \mathbb{P}\left(X \wedge Y\left(\frac{1}{4}\lambda\right)\right) =$$

$$= \mathbb{P}\left(X | Y\left(\frac{1}{4}\lambda\right)\right) \cdot \mathbb{P}\left(Y\left(\frac{1}{4}\lambda\right)\right) \geq$$

$$\geq \left(1 - \left(\frac{1}{2}\right)^{\frac{1}{4}\lambda}\right) \cdot (1 - \mu_1(\lambda)) = 1 - \mu_2(\lambda)$$

where μ_1 is negligible (it can be easily proven using the Stirling's approximation) so also μ_2 is negligible. That ends the proof that the Buyer can be cheated with probability at most $\left(\frac{b}{a}\right)^b$ + negligible.

Assume now that the Seller is honest and the Buyer is not. Observe first, that the Seller reveals signatures $\hat{\sigma}^i$ only when she is redeeming the transaction T_1. She does it only after T_1 is included into the blockchain

so T_1 cannot be withdrawn. Additionally at that moment only the Seller can redeem T_1. That is true, because from the lemma 1 we know that the Buyer cannot sign any message with respect to any of the public keys $\hat{pk}^i$ until τ_0 seconds elapsed. He needs to create such a signature to redeem T_1. The Seller broadcasts T_2 much earlier than after τ_0 seconds so it will be included into the blockchain so she will gain d ₿.

What remains to be proven is that the Buyer learns no information about x until the Seller reveals signatures $\hat{\sigma}^i$ by broadcasting T_2. But we know that the $\mathcal{F}$ is a zero knowledge proof and for each $\mathsf{Setup}^{i,j}$ phase the Buyer sends at most one challenge $c_B^{i,j}$ and learns at most one response $r^{i,j}$. This response corresponds the the challenge $c_B^{i,j}$, so it gives him no information about x. The only other values that the Buyer learns are either commitments or values encrypted with unknown for him keys. But we assumed that the encryption schemes and commitment schemes are secure, so except with the negligible probability the Buyer learns no information about x if the Seller does not get the Buyer's funds. □

E Proof of Lemma 3

We have to prove that this protocol satisfy completeness, validity and zero knowledge properties.

The completeness is easy to see. If the Prover knows the factorization of $n = pq$ then he can use the Tonelli-Shanks algorithm [42] to find both square roots of any quadratic residue $\bmod p$ and q and by the Chinese remainder theorem he can find all four square roots $\bmod n$. Two of them are smaller than $\frac{n}{2}$ so he will be able to find and open the commitment for $r_b = x$.

Now we will prove validity. Let $\hat{P}$ be a (possibly malicious) Prover that convinces the honest Prover with probability α. That means, that with probability at least α he commits to r where r is uniformly chosen square root of y.

We will now define the extractor E that has access to $\hat{P}$ and controls the random oracle. E will play a role of the honest Verifier in the interaction with $\hat{P}$. So at the beginning E samples random x and sends $y = x^2 \bmod n$ to $\hat{P}$. Then $\hat{P}$ makes a number of queries to the random oracle and sends two commitments to E. If $\hat{P}$ made a query to the random oracle on value x' s.t. x' is a square root of y different than x and $n - x$ then the extractors calculates and returns a factor of n by computing $p = \gcd(n, x - x')$. Otherwise the extractor aborts.

The probability that $\hat{P}$ is able to open a commitment without first querying the random oracle is negligible, so with probability at least α minus negligible $\hat{P}$ will query the oracle on the other square root of y so with probability at least α minus negligible the Extractor will be able to return the factorization of n.

To prove the zero knowledge property we will construct a simulator S that has access to a (possibly malicious) verifier $\hat{V}$ and controls the random oracle. S will simulate the interaction between $\hat{V}$ and the honest verifier. When S is given value y from $\hat{V}$ then it returns as the commitments two randomly sampled strings s_0, s_1 of a proper length. After receiving x the Simulator aborts if $x^2 \neq y \bmod n$ or $x > \frac{n}{2}$. Otherwise he randomly samples $b \leftarrow \{0, 1\}$, opens s_b as a commitment to x and programs the random oracle s.t. the commitment of x is exactly s_b. Because of that the verifier $\hat{V}$ will accept the opening of the commitment so he will not be able to distinguish if he is interacting with S or the honest Prover. □

F Sketch of the security of the protocol for trading across chains

Because of lack of space we do not present here a formal security definition or a formal security proof. We just argument how the parties can cheat and why the probability of stealing the other party's funds is small.

Observe first, that the public keys created for Alice and for Bob will be uniformly random and the secret key will be indeed secret-shared between the parties. This claim is clear for key pairs created for Bob as they are created in the USG protocol. And each key pair created for Alice is equal to $d_A^{k_j} \cdot d_B^{l_j} \cdot g$ where $d_B^{l_j}$ was first chosen by Bob and $d_A^{k_j}$ was later chosen independently (because of the commitments) by Alice.

Assume first that Bob is honest and Alice is not. Alice can cheat only in the USG protocol, including the creation of the values x^i. But as in the SellWitness protocol Alice would have to cheat in exactly b iterations of the KSignGen procedure and Bob would have to choose exactly these invalid positions to get cheated. That is true, because he needs only one secret key generated for him to redeem transaction T_B^1 and get his money back and only one secret key generated for Alice to get her funds from transaction T_A^1. So the probability that Alice will cheat Bob is at most $\left(\frac{b}{a}\right)^b$.

If Alice is honest and Bob is not, then the only place when Bob can cheat and will not be caught immediately is when calculating timed commitments to d_B^i. But if at least one of these timed commitments were computed honestly then Alice will be able to open it and get her funds back (in case when e.g. she broadcasts T_A^1 but Bob do not broadcasts T_B^1) because she needs only one signature to redeem T_A^1. And the probability that Bob will cheat in all b cases and in none of the others (otherwise Alice would notice) is at most $\left(\frac{b}{a}\right)^b$.

Observe also that the malleability of the transactions is not a problem. The only case when it can matter is the case of the transaction T_B^1 — if it is mauled then Alice will not be able to use signatures σ^i and they will have to restart the protocol.

Summarising, none of the parties can be cheated except with the probability $(b/a)^b$ + negligible.

G Protocol performance graphs

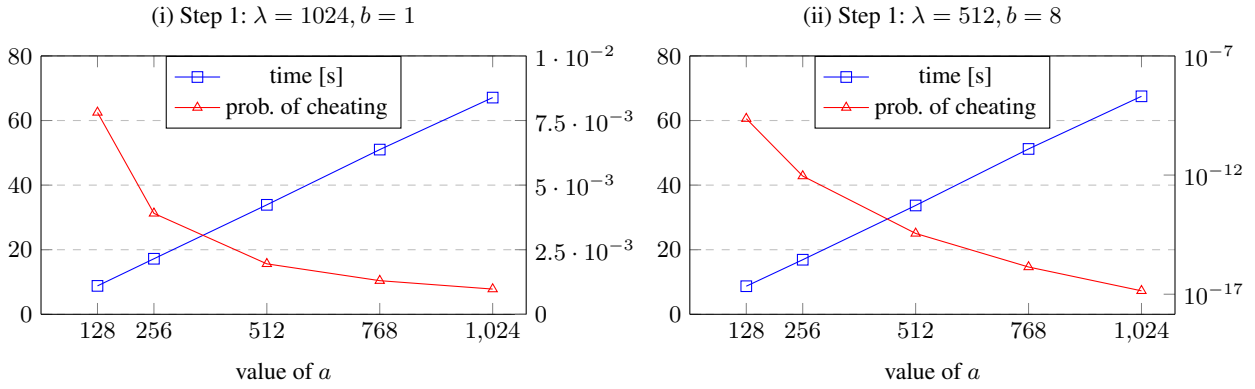

Fig. 7: The running time of the Step 1 and the probability that the Seller successfully cheats the Buyer in the Step 1 of the SellWitness protocol for the following fixed parameters: (i) $\lambda = 1024$ and $b = 1$ (i.e. using only standard single-signature transactions), and (ii) $\lambda = 512$ and $b = 8$ (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.

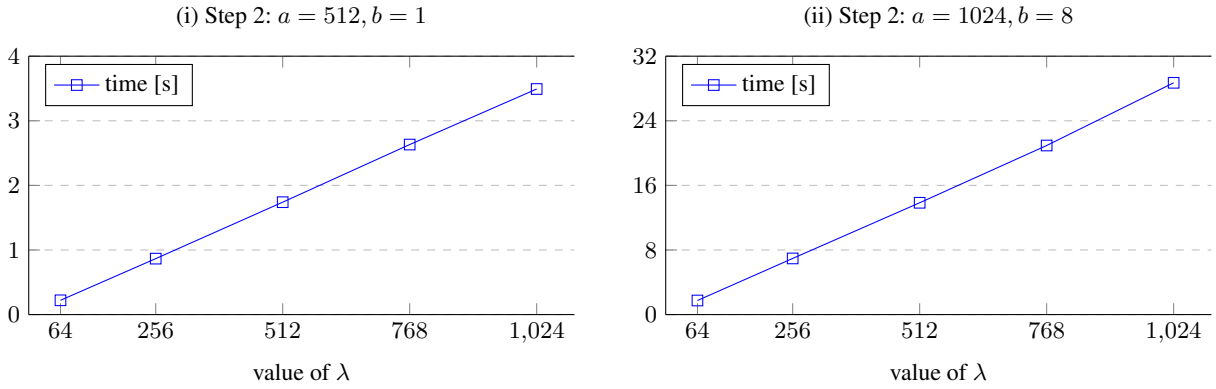

Fig. 8: The running time of the Step 2 of the SellWitness protocol for the following fixed parameter: (i) $a = 512$ and $b = 1$ (i.e. using only standard single-signature transactions), and (ii) $a = 1024$, and $b = 8$ (i.e. using multi-signature transactions with the greatest parameters that are standard 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.

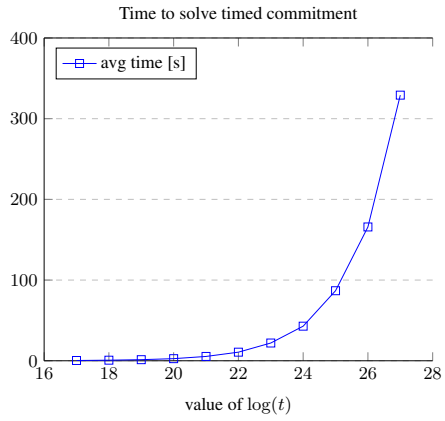

Fig. 9: The running time of an algorithm solving a timed commitment for different difficulties t.