Liar, Liar, Coins on Fire!

Penalizing Equivocation by Loss of Bitcoins

Tim Ruffing
CISPA, Saarland University
tim.ruffing@mmci.uni-saarland.de

Aniket Kate*
Purdue University
aniket@purdue.edu

Dominique Schröder
CISPA, Saarland University
schroeder@cs.uni-saarland.de

ABSTRACT

We show that equivocation, i.e., making conflicting statements to others in a distributed protocol, can be monetarily disincentivized by the use of crypto-currencies such as Bitcoin. To this end, we design completely decentralized *non-equivocation contracts*, which make it possible to penalize an equivocating party by the loss of its money. At the core of these contracts, there is a novel cryptographic primitive called *accountable assertions*, which reveals the party's Bitcoin credentials if it equivocates.

Non-equivocation contracts are particularly useful for distributed systems that employ public append-only logs to protect data integrity, e.g., in cloud storage and social networks. Moreover, as double-spending in Bitcoin is a special case of equivocation, the contracts enable us to design a payment protocol that allows a payee to receive funds at several unsynchronized points of sale, while being able to penalize a double-spending payer after the fact.

Categories and Subject Descriptors

C2.4 [**Computer-communication networks**]: Distributed systems; K4.4 [**Computers and society**]: Electronic commerce—*cybercash, digital cash, payment schemes, security*

Keywords

crypto-currencies; Bitcoin; equivocation; append-only logs; accountability; double-spending; payment channels

1. INTRODUCTION

Making conflicting statements to others, or *equivocation*, is a simple yet remarkably powerful tool of malicious participants in distributed systems of all kinds [4, 19, 20, 33]. In distributed computing protocols, equivocation leads to Byzantine faults and fairness issues. When feasible, equivocation is handled by assuming an honest majority (i.e., larger replication factors [27]), synchrony assumptions and digital signatures [26], or trusted hardware [4, 19, 20, 33]. Moreover, publicly verifiable append-only logs [22, 23, 24, 35] make it possible to detect equivocation after the fact but they do not suffice to stop or penalize equivocation.

Decentralized crypto-currency systems such as Bitcoin [10, 36] and its derivatives follow a novel approach to handle equivocation. To protect against equivocation in the form of *double-spending*, i.e., spending the same funds to different parties, Bitcoin employs a special decentralized public append-only log based on proof of work called the *blockchain*: In a decentralized crypto-currency, users transfer their funds by publishing digitally signed transactions. Transactions are confirmed only when they are included in the blockchain, which is generated by currency miners that solve proof-of-work puzzles. Although a malicious owner can sign over the same funds to multiple receivers through multiple transactions, eventually only one transaction will be approved and added to the publicly verifiable blockchain.

As a result, to stop equivocation, it is possible to record all messages in a distributed system that are vulnerable to equivocation in a blockchain. Nevertheless, due to proof-of-work computations and the decentralized nature of blockchain systems, the process of reaching consensus is not only expensive but also only slowly converging. In Bitcoin, it takes tens of minutes to reach consensus on the set of valid transactions.

To enable transactions be performed faster, a contractual solution in the form of payment channels [45, 48] is emerging in the Bitcoin community [39, 47]. Here, a payer makes a time-locked deposit for his predetermined payee such that double-spending (or equivocation) is excluded even when payments are performed offline and without waiting. However, payment channels are not secure against double-spending when the payee runs several geographically distributed and unsynchronized points of sale, e.g., a bus company selling tickets on buses with only sporadic Internet connectivity.

Our goal in this paper is to address these equivocation issues by a generic solution that disincentives paltering and is applicable to various distributed systems and scenarios including the aforementioned payment channels with unsynchronized points of sale.

1.1 Contributions

Our key idea towards preventing equivocation is to use Bitcoin to prescribe a monetary penalty for equivocation.

Accountable Assertions. As a first step, we establish a cryptographic connection between equivocation and the loss of funds by introducing a cryptographic primitive called

*The work was done while the author was still at Saarland University.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813686.

accountable assertions (Section 4). The main idea of this primitive is to bind *statements* to *contexts* in an accountable way: if the attacker equivocates, i.e., asserts two contradicting statements in the same context, then any observer can extract the attacker's Bitcoin secret key and, as a result, use it to force the loss of the attacker's funds.

We present a construction of accountable assertions based on chameleon hash functions [30] and prove it secure in the random oracle model under the discrete logarithm assumption (Section 5). A performance evaluation of our construction demonstrates its practicality with respect to computation, communication, and storage costs.

Non-equivocation Contracts. To ensure that a secret key obtained through equivocation is indeed associated with funds, every party that should be prevented from equivocating is required to put aside a certain amount of funds in a *deposit* [2, 6, 32, 40]. These funds are *time-locked* in the deposit, i.e., the depositor cannot withdraw them during a predetermined time period. This prevents an attacker from spending the funds and thus rendering the secret key useless just before equivocating.

Accountable assertions and deposits together enable us to design *non-equivocation contracts*, a generic method to penalize paltering in distributed systems (Section 6). We propose several applications of non-equivocation contracts to ensure the linearity of append-only logs [22, 23, 24, 35].

Asynchronous Payment Channels. Bitcoin *payment channels* protocols [45, 48] enable a user to perform payments to a predetermined party offline and without waiting for the consensus process.

However, if a payee is a distributed entity (e.g., a bus service with several buses as points of sale with only sporadic Internet connectivity) then even payment channels do not prevent double-spending. Since double-spending is an instance of equivocation, non-equivocation contracts enable us to design *asynchronous payment channels*, which make it possible to penalize double-spending payers (Section 7).

Double-Authentication-Preventing Signatures. Of independent interest, we observe that accountable assertions are similar to *double-authentication-preventing signatures* (DAPS) as proposed by Poettering and Stebila [38]. While accountable assertions are in general a weaker primitive, certain accountable assertions are DAPS. It was left as an open problem to construct DAPS based on trees or chameleon hash functions [38]. We solve these problems, and our accountable assertion scheme based on Merkle tress *and* chameleon hash functions in the random oracle model yields the first DAPS scheme secure under the discrete logarithm assumption (Appendix A). For practical parameters, it is two orders of magnitude faster than the original DAPS construction [38], and uses one order of magnitude less communication.

2. OVERVIEW

We conceptualize decentralized non-equivocation contracts and discuss their potential applications.

Problem Statement. Equivocation, i.e., making conflicting statements to different protocol parties, is a universal problem in malicious fault-tolerant security protocols involving three or more parties [4, 19, 20, 33]. In all bounded or partial synchronous communication settings, equivocation can be detected using digital signatures (together with a public-key infrastructure) and some interaction among the parties [20]: two recipients who are expected to receive the same message from a sender can exchange the received signed messages to expose and prove equivocation. This principle underlies many append-only logs [22, 23, 24, 35].

However, it is often not possible to impose a penalty on the maliciously or carelessly equivocating sender after the fact, as the sender may be anonymous or pseudonymous. Even when the sender is not anonymous and may lose her reputation once a case of equivocation is detected, the effect of such paltering on the recipient can be damaging.

Key Idea. Our key idea is to let the sender create a time-locked Bitcoin deposit [2, 6, 32, 40] that can be opened by the recipients if the sender equivocates. In case of an equivocation, the funds will be given either to a predefined beneficiary or, once the expiry time of the deposit is reached, to the miners. If the expected loss is high enough, the attacker has no incentive to make conflicting statements.

Threat Model. The attacker is a malicious sender whose goal is to equivocate without losing the deposit. To achieve that goal, the attacker can deviate arbitrarily from the prescribed protocol but she does not risk to lose her deposit if the expected loss is higher than the expected gain.

We assume that the attacker cannot break the fundamental security of Bitcoin, e.g., the attacker does not have the majority of computing power in the Bitcoin network.

Non-equivocation Contracts. We describe the main idea of non-equivocation contracts, which are a form of smart contracts [11, 29, 40], in more detail. The sender A creates a time-locked deposit as a guarantee for her honest behavior. The deposit is secured by the sender's secret key sk_A; the corresponding public key is pk_A. Furthermore, the deposit expires at some point T in the future. That is, even though A owns the secret key sk_A, she cannot access the funds in the deposit until time T. Before time T, only A *together* with a predefined beneficiary P can access the funds. This beneficiary will be given the funds if A equivocates. (There is also a variant of deposits for which the beneficiary is a randomly selected miner. We will explain this later.)

Once the deposit is confirmed by the Bitcoin network, parties are ready to receive statements from the sender A.

Non-equivocating contracts are built on the idea that it is possible to learn the key sk_A if the sender A equivocates. To enforce this cryptographically, we introduce *accountable assertions*, which allow the user A to produce assertions τ of statements st in contexts ct (where st and ct can be arbitrary bitstrings) under the public key pk_A.

The sender A is held accountable in the following sense: If A behaves honestly, sk_A will stay secret, and A can use it to withdraw the deposit once time T has been reached. However, if A equivocates to some honest users B and C, i.e., A asserts two different statements $st_0 \neq st_1$ in the same context ct, then B and C can use st_0, st_1, ct and the two corresponding assertions τ_0 and τ_1 to extract the sender's secret key sk_A. Due to the way the deposit is created, the recipients B and C alone cannot make use of sk_A. However, B and C can send sk_A to the beneficiary P, who can use sk_A together with his credentials to withdraw the deposit and thereby penalize the malicious sender A.

Note that B, C and P could as well be protocol parties that belong to essentially the same distributed entity but are just not synchronized when receiving statements from A.

3. BACKGROUND ON BITCOIN

Bitcoin is an online digital cryptographic currency run by a decentralized peer-to-peer network. In this section, we explain the basics of Bitcoin that are relevant to our work. For a detailed explanation of the mechanics of Bitcoin, we refer the reader to the Bitcoin developer guide [7].

A user in the Bitcoin network is identified using one or more pseudonymous addresses. Technically, an address is the hash of a public key *pk* of the ECDSA signature scheme such that the owner of the corresponding secret key *sk* can use it to transfer bitcoins (symbol: ₿) associated with her address to another address by signing *transactions*.

Blockchain. Miners include transactions in *blocks*. By solving a proof-of-work (POW) puzzle, a block including its transactions is added to the *blockchain*. Once added, a block and its transactions are difficult to modify because blocks are cryptographically chained together, and modifying a block involves re-doing the POW for this and all sequential blocks. A transaction that has been included in the blockchain and backed up by the POW computations of several blocks is thus difficult to invalidate. Most users consider a transaction confirmed if it has has been backed up by least six blocks, and the Bitcoin network takes 10 min on average to perform the POW of one block.

Scripts. Bitcoin employs a scripting language to specify under which conditions an *unspent output*, i.e., some unspent funds in the blockchain, can be spent. The language is a simple stack-based language. It is intentionally not Turing-complete to avoid complexity and the possibility of creating scripts that are expensive to execute, and could consequently lead to denial-of-service attacks, because every node in the Bitcoin network must execute them.

Each transaction sends funds to a script (called *ScriptPubKey*), i.e., a small program that specifies the conditions that must be fulfilled to spend the funds. To spend the funds, the spender must provide an initial execution stack with input values. The transaction is valid if the script terminates successfully on this initial stack.

3.1 Deposits

Using specially-crafted scripts, funds can be locked away in a so-called *deposit*, where they can only be accessed under a set of predetermined conditions. While scripts can express a variety of such conditions [7], we focus on *time-locked* deposits with the property that the depositor cannot access the funds in the deposit until a specified *expiry time*.

With non-equivocation contracts in mind, we consider two types of deposits that differ in the *beneficiary*, i.e., the party that receives the funds in case of equivocation. The deposits of the first type do not specify a beneficiary. In this case, the beneficiary will be a randomly selected miner. Deposits of the second type are associated with an *explicitly* beneficiary P identified by his public key pk_P.

Creating Deposits. To create time-locked deposits, we use an upcoming feature [17] in Bitcoin, which introduces a new script command denoted by `CHECKLOCKTIMEVERIFY` [48].[1] This command takes one argument T, the *expiry time*, from the execution stack and compares it to the *nLockTime* data field of the transaction. If $nLockTime < T$, the evaluation fails and the transaction is consequently invalid. Thus, only transactions with $nLockTime \geq T$ can spend the funds covered by such a script. By the semantics of *nLockTime*, those transactions are valid only in blocks found after time T, and consequently, the funds protected by `CHECKLOCKTIMEVERIFY` are spendable only after T.

We remark that the value of *nLockTime* can be specified either by a UNIX timestamp or a *height of a block*, which is the number of blocks that precede it in the blockchain. Throughout the paper, we use timestamps, and to simplify presentation, we ignore that miners have some flexibility to lie about the current time [8]; an safety-margin of at least 120 min must be added to T to account for that issue.

Deposits Without Explicit Beneficiary. Suppose that some user A wishes to create a deposit with expiration time T without an explicit beneficiary. Then, A sends the desired amount ₿ d to the following script:

$(T + T_{conf}^{impl})$ `CHECKLOCKTIMEVERIFY DROP`
pk_A `CHECKSIG`

The literals $(T + T_{conf}^{impl})$ and pk_A in the script denote push operations that push a constant value on the stack. The value T_{conf}^{impl} is a safety margin; we postpone its discussion to the analysis of non-equivocation contracts (Section 6.1).

The first line of the script ensures that the deposit cannot be spent before time T as explained. (`DROP` just drops the constant value from the stack.) In the second line, `CHECKSIG` takes the key pk_A and a signature σ from the stack; σ is supposed to be provided by the spender on the initial stack. `CHECKSIGVERIFY` verifies that σ is a valid signature of the spending transaction under the key pk_A, pushing the boolean result of the verification on the stack. This boolean value is the output of the script. Thus, if the check succeeds, the transaction is valid; otherwise it is invalid. In sum, the script ensures that the funds can only be spent after T and only by a transaction signed under pk_A.

If the corresponding secret key sk_A is revealed, everybody can create transactions that try to spend the funds from time $(T + T_{conf}^{impl})$ on. Whenever this happens, each miner has a large incentive to include a transaction in a block that sends the money to him. Consequently, the miner that finds the next block will claim the funds.

Deposits with Explicit Beneficiary. Suppose that a user A wishes to create a deposit with an explicit beneficiary P. Then, A sends the desired amount ₿ d to the following script:

`IF`
    pk_P `CHECKSIGVERIFY`
`ELSE`
    $(T + T_{conf}^{expl} + T_{net}^{expl})$ `CHECKLOCKTIMEVERIFY DROP`
`ENDIF`
pk_A `CHECKSIG`

In this script T_{conf}^{expl} and T_{net}^{expl} are safety margins, which will be discussed below. `IF` obtains its condition from the stack, allowing the spender to choose the branch to be executed. `CHECKSIGVERIFY` is like `CHECKSIG` but causes the whole script to fail immediately if the signature is not valid (instead of pushing the result of the signature verification to the stack).

This script ensures that before time T, the funds can be spent only if the spending transaction is signed under both pk_A and pk_P. Thus, if P learns sk_A before time T, he can spend the funds. Otherwise, A is refunded after time $(T + T_{conf}^{expl} + T_{net}^{expl})$, even if P disappears.

[1] Deposits with explicit beneficiaries and payment channels as described in the following paragraphs are possible even without `CHECKLOCKTIMEVERIFY` [1, 40, 45].

The safety margins are necessary because the closing transaction must have been broadcast to the Bitcoin network and confirmed by it before the deposit can be spent by A alone. For the broadcast, $T^{expl}_{net} = 10\,\text{min}$ is more than sufficient [21]. For the confirmations, we except the network to find 24 blocks in $T^{expl}_{conf} = 240\,\text{min}$. Since their arrival is Poisson-distributed, the probability that fewer than six desired blocks have been found is $\Pr[X \leq 5] < 2^{-18}$ for $X \sim \text{Pois}(24)$.

3.2 Payment Channels

Payment channels [45, 48] allow a user A to perform many transactions to a predefined recipient B up to a predefined amount ₿ d of money. Although establishing a channel between A and B involves waiting for a transaction to be confirmed, the advantages of a payment channels are various: First, no matter how many payments are sent, only two transactions have to be included in the blockchain, namely one to establish and one to close the channel. This makes payment channels a promising method to scale the Bitcoin network to many more transactions [39, 47]. Second, A can perform payments to B even if both parties are offline, once the channel has been established. Third, fast transactions are possible through the payment channel, because B does not have to wait for the transaction to be confirmed.

Creating a Payment Channel. To create a payment channel from A to B with maximal payment value ₿ d and expiry time T, A follows the procedure for creating a deposit with explicit beneficiary B.

B waits until the deposit is confirmed by the Bitcoin network. From now on, the funds can only be spent if both A and B agree because any spending transaction must be signed by both A and B to be valid. Since B will only endorse transactions that send funds to him, B is protected from attempts by A to send funds to another party (or back to herself), i.e., B is protected from double-spending attempts.

Paying Through the Channel. The channel has an associated state b that specifies how many of the ₿ d have been paid so far to B. In the beginning, $b = 0$, i.e., all money in the channel belongs to A and none belongs to B. To pay through the channel, i.e., to raise b to b', A creates an ordinary Bitcoin transaction that sends ₿ b' fron the deposit to B. She signs this transaction with her secret key sk_A, and sends the transaction to B, who validates the transaction and the correctness of the signature. However, the transaction is not yet signed by B or published to the Bitcoin network.

Closing the Channel. The channel has to be closed before time T. If B wants to close the channel at some state $\hat{b}$, he sends the most recently received transaction, i.e., the one with the value $\hat{b}$, to the Bitcoin network. Once the network confirms the transaction, B has received ₿ $\hat{b}$.

If B does not close the channel by time T, e.g., as B has disappeared, A can claim the whole channel of value ₿ d.

4. ACCOUNTABLE ASSERTIONS

In this section we introduce accountable assertions. Intuitively, this primitive allows users to assert *statements* in *contexts* such that users can be held accountable for equivocation: On the one hand, if the user asserts two different statements $st_0 \neq st_1$ in the same context ct, then a public algorithm can *extract* the secret key ask of the user from the two assertions. On the other hand, secrecy of the secret key ask remains intact for a well-behaved non-equivocating user.

Accountable assertions are supposed to be attached to other public-key primitives, i.e., the key pairs are supposed to correspond to key pairs of the other primitive. For example, the key pairs of our scheme will be valid ECDSA key pairs as used in Bitcoin. Attaching accountable assertions to other primitives is crucial in practice because the concrete secret key used in accountable assertions needs to be worth something, e.g., for redeeming funds. Otherwise, the user has no incentive to keep it secret in the first place.

Definition 1 (Accountable Assertions). *An accountable assertion scheme Π is a tuple of ppt algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Assert}, \mathsf{Verify}, \mathsf{Extract})$ *as follows:*

- $(apk, ask, auxsk) \leftarrow \mathsf{Gen}(1^\lambda)$*: The* key generation *algorithm outputs a key pair consisting of a public key apk and a secret key ask, and auxiliary secret information auxsk. It is required that for each public key, there is exactly one secret key, i.e., for all λ and all outputs (apk, ask) and (apk', ask') of $\mathsf{Gen}(1^\lambda)$ with $apk = apk'$, we have $ask = ask'$.*
- $\tau/\bot \leftarrow \mathsf{Assert}(ask, auxsk, ct, st)$*: The stateful* assertion algorithm *takes as input a secret key ask, auxiliary secret information auxsk, a context ct, and a statement st. It returns either an assertion τ or $\bot$ to indicate failure.*
- $b \leftarrow \mathsf{Verify}(apk, ct, st, \tau)$*: The* verification algorithm *outputs* 1 *if and only if τ is a valid assertion of a statement ct in the context st under the public key apk.*
- $ask/\bot \leftarrow \mathsf{Extract}(apk, ct, st_0, st_1, \tau_0, \tau_1)$*: The* extraction algorithm *takes as input a public key apk, a context ct, two statements st_0, st_1, and two assertions τ_0, τ_1. It outputs either the secret key ask or $\bot$ to indicate failure.*

The accountable assertion scheme Π is correct *if for all security parameters λ, all keys $(apk, ask, auxsk) \leftarrow \mathsf{Gen}(1^\lambda)$, all statements st, all contexts ct, and all assertions $\tau \neq \bot$ resulting from a successful assertion $\tau \leftarrow \mathsf{Assert}(ask, auxsk, ct, st)$, we have $\mathsf{Verify}(apk, ct, st, \tau) = 1$.*

In case of equivocation, only the secret key ask will be guaranteed to be extractable, but not the auxiliary secret information $auxsk$.

Completeness. Our definition of accountable assertions allows the assertion algorithm to fail. We do not consider such failure a problem if it happens only with small (but not necessarily negligible) probability, because failure hurts only the liveness of the system that makes use of the accountable assertions. However, liveness is typically not guaranteed anyway due to unreliable networks. As a consequence, we do not insist generally on accountable assertions fulfilling a completeness criterion.

At first glance, this might look a bit contrived, but the purpose of this is to trade off reliability against efficiency. Accountable assertions are, unlike signatures, not required to be unforgeable, and it turns out that setting unforgeability aside will enable a more efficient construction.

To understand how failing and unforgeability are related, suppose an attacker asks a user to assert a statement st_0 in a context ct_0, i.e., to output $\tau_0 \leftarrow \mathsf{Assert}(ask, auxsk, ct_0, st_0)$. Due to the lack of unforgeability, the attacker might use τ_0 to obtain another assertion τ_1 that is valid for some *related* but different context $ct_1 \neq ct_0$ and the same statement st_0. So far, this is not a problem: the attacker cannot use the extraction algorithm to obtain the secret key ask from τ_0 and τ_1, because the two assertions are valid in different contexts $ct_0 \neq ct_1$. However, the attacker can now ask the user to assert another statement $st_1 \neq st_0$ in the context

ct_1, i.e., to output $\tau_1' \leftarrow \mathsf{Assert}(ask, auxsk, ct_1, st_1)$. Observe that this is a valid request: the attacker does not ask the user to equivocate because the user has not asserted any statement in the context ct_1 so far. But if the user replied to the request, the attacker could run the extraction algorithm $\mathsf{Extract}(apk, ct_1, st_1, st_1', \tau_1, \tau_1')$ to extract the secret key ask.

To avoid this attack, while allowing for a construction that is "forgeable" as just described, the stateful assertion algorithm may fail if it detects that the context ct_1, for which an assertion is requested, is related to a previously used context ct_0.

Nevertheless, the ability of the attacker to force failure may be a problem in certain scenarios, e.g., if it allows the attacker to perform a denial-of-service attack. In those cases, it is possible to consider *complete* accountable assertions, which are guaranteed to succeed on all honestly chosen inputs.

Definition 2 (Completeness). *An accountable assertion scheme* $\Pi = (\mathsf{Gen}, \mathsf{Assert}, \mathsf{Verify}, \mathsf{Extract})$ *is* complete *if for all security parameters* λ*, all key pairs* (apk, ask) *output by* $\mathsf{Gen}(1^\lambda)$*, all statements* st*, and all contexts* ct*, we have* $\mathsf{Assert}(ask, auxsk, ct, st) \neq \bot$.

Note that the definition of accountable assertions additionally demands correctness whenever $\mathsf{Assert}(ask, auxsk, ct, st) \neq \bot$.

4.1 Security of Accountable Assertions

Accountable assertions need to fulfill two security properties. The first security property is *extractability*, which states that whenever two distinct statements have been asserted in the same context, the secret key can be extracted.

Definition 3 (Extractability). *An accountable assertion scheme* $\Pi = (\mathsf{Gen}, \mathsf{Assert}, \mathsf{Verify}, \mathsf{Extract})$ *is* extractable *if for all ppt attackers* $\mathcal{A}$,

$$\begin{aligned}\Pr[&\mathsf{Extract}(apk, ct, st_0, st_1, \tau_0, \tau_1) \neq ask \\ &\wedge\ \forall b \in \{0,1\}, \mathsf{Verify}(apk, ct, st_b, \tau_b) = 1 \\ &\wedge\ st_0 \neq st_1 : (apk, ct, st_0, st_1, \tau_0, \tau_1) \leftarrow \mathcal{A}(1^\lambda)]\end{aligned}$$

is negligible in λ*. Here,* ask *is the unique secret key corresponding to* apk*.*

The second security property *secrecy* is opposed to extractability. Secrecy prevents the extraction of the secret key against an attacker who can ask the challenger to assert chosen statements in chosen contexts. Since accountable assertions are extractable, the attacker's success is excluded after requesting the assertion of two different statements in the same context.

Definition 4 (Secrecy). *An accountable assertion scheme* $\Pi = (\mathsf{Gen}, \mathsf{Assert}, \mathsf{Verify}, \mathsf{Extract})$ *is* secret *if for all ppt attackers* $\mathcal{A}$*, the probability that the experiment* $\mathsf{Sec}^{\Pi}_{\mathcal{A}}(\lambda)$ *returns* 1 *is negligible in* λ*, where the experiment* $\mathsf{Sec}^{\Pi}_{\mathcal{A}}(\lambda)$ *is defined as follows.*

Experiment $\mathsf{Sec}^{\Pi}_{\mathcal{A}}(\lambda)$
  $(apk, ask, auxsk) \leftarrow \mathsf{Gen}(1^\lambda)$
  $Q := \emptyset$
  $ask^* \leftarrow \mathcal{A}^{\mathsf{Assert}''(ask, auxsk, \cdot, \cdot)}(apk)$
  return 1 *iff* $ask^* = ask$
    $\wedge\ (\nexists ct, st_0, st_1.\ st_0 \neq st_1 \wedge \{(ct, st_0), (ct, st_1)\} \subseteq Q)$
Oracle $\mathsf{Assert}''(ask, auxsk, ct, st)$
  $Q := Q \cup \{(ct, st)\}$
  return $\mathsf{Assert}(ask, ask, ct, st)$

Limitations of the Secrecy Definition. Recall that a secret key used with accountable assertions must be worth something, e.g., a valid ECDSA secret key that protects funds in Bitcoin. We would like to draw the reader's attention to the fact that the definition of secrecy does not take into account the other usages of the secret key. That is, while our secrecy definition of accountable assertions is meaningful in itself, it is only a heuristic for analyzing their security when combined with other primitives, and it is formally not guaranteed that the use of secret accountable assertions keeps the security of the other primitives intact.[2] While we are confident that the combined use of our accountable assertions construction (Section 5) together with ECDSA does not render ECDSA insecure in practice, a more formal treatment of the composability of accountable assertions with other properties is desirable. We leave this for future work.

Relation to DAPS. Double-authentication-preventing signatures (DAPS) [38] have similar properties as accountable assertions, but are additionally required to be unforgeable. We have discussed an informal relation between the unforgeability of accountable assertions and their completeness. This intuition can be formalized, and it turns out that a slightly modified variant of our accountable assertions construction (Section 5) and is an efficient DAPS scheme. We refer the reader to Appendix A for a discussion.

5. CONSTRUCTION AND ANALYSIS

In this section, we propose a construction of accountable assertions based on chameleon hash functions. Our construction builds upon the idea of chameleon authentication trees (CATs), as suggested by Schröder and Schröder [42] and improved in followup schemes [31, 43]. As opposed to these schemes, the novelty of our construction is the extractability.

Chameleon Hashes. A chameleon hash function is a randomized hash function that is collision-resistant but provides a trapdoor to efficiently compute collisions [30]. Formally, a chameleon hash function $\mathcal{CH}$ is a tuple of ppt algorithms $(\mathsf{GenCh}, \mathsf{Ch}, \mathsf{Col})$. The key generation algorithm $\mathsf{GenCh}(1^\lambda)$ returns a key pair (cpk, csk) consisting of a public key cpk and a trapdoor csk. The evaluation function $\mathsf{Ch}(cpk, x; r)$ produces a hash value for a message x and a random value r; we typically write just $\mathsf{Ch}(x; r)$ when cpk is clear from the context. The collision-finding algorithm $\mathsf{Col}(csk, x_0, r_0, x_1)$ takes as input a trapdoor csk and a triple (x_0, r_0, x_1); it outputs some value r_1 such that $\mathsf{Ch}(x_0; r_0) = \mathsf{Ch}(x_1; r_1)$. Chameleon hash functions need to fulfill collision-resistance and uniformity as defined by Krawczyk and Rabin [30].

In addition to these standard security properties, we require the trapdoor to be extractable from a collision. While this extractability is typically considered a problem [3, 18, 42], it turns out to be a crucial requirement for our construction.

Definition 5 (Chameleon Hash Extractability). *A chameleon hash function* $\mathcal{CH} = (\mathsf{GenCh}, \mathsf{Ch}, \mathsf{Col})$ *with unique keys is* extractable *if there exists a polynomial-time algorithm* $\mathsf{ExtractCsk}$ *with the following property: For all key pairs* (cpk, csk) *output by* GenCh*, and for all input pairs*

[2]Indeed, given an unforgeable signature scheme and a secret accountable assertion scheme, one can construct a pathological unforgeable signature scheme that is insecure when $f(ask)$ is leaked for a one-way function f, and one can construct a secure accountable assertion scheme that leaks $f(ask)$.

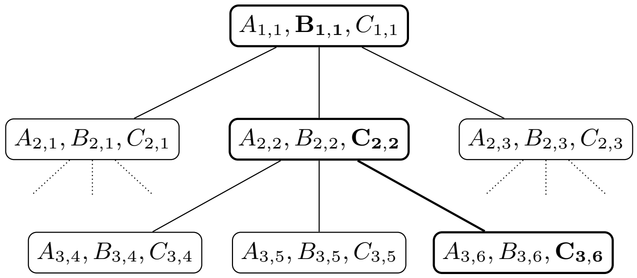

Figure 1: A tree as in our construction

(x_0, r_0) *and* (x_1, r_1) *with* $x_0 \neq x_1$ *and* $\mathsf{Ch}(x_0; r_0) = \mathsf{Ch}(x_1; r_1)$, *we have* $\mathsf{ExtractCsk}(cpk, x_0, r_0, x_1, r_1) = csk$.

5.1 Intuition

First Approach. One obvious but flawed approach to construct accountable assertions is to let the assertion algorithm output a random value r such that $ct = \mathsf{Ch}(st; r)$. The intuition is that if the attacker does this for two different statements st_0, st_1 in the same context ct, then this would yield a collision $\mathsf{Ch}(st_0; r_0) = ct = \mathsf{Ch}(st_1; r_1)$ in the chameleon hash function, and one could extract the trapdoor. This simple idea does not work: ct would live in the output space of the chameleon hash function, but in most of the chameleon hash functions, the trapdoor can only be used to find collisions efficiently, not to invert the function.

Full Idea. Observe that the aforementioned approach works, however, as a scheme that supports only one assertion in an arbitrary but fixed context, for which inverting the chameleon hash is not necessary. If the public key of the accountable assertions scheme includes $\mathsf{Ch}(x^*; r^*)$ for randomly chosen x^* and r^*, then one can use the trapdoor to compute r as an assertion for a statement st such that $\mathsf{Ch}(x^*; r^*) = \mathsf{Ch}(st; r)$.

The basic idea of our construction is to generalize this approach to many contexts by applying it recursively, resulting in a Merkle-style tree based on chameleon hash functions. The contexts are associated with the leafs of the tree, and a digest of the root node is part of the public key.

Let n denote the arity and ℓ denote the depth of the tree. We explain the main steps with the help of Fig. 1 for $n = 3$. In our construction (a digest of) the context defines its position in the tree. That is, the context with the lowest digest is stored in the leftmost leaf and the context with the highest digest in the rightmost node. Since the tree is of exponential size, storing or computing the entire tree at once is not possible. Instead, we compute each element $A_{i,j}, B_{i,j}, C_{i,j}$ as a chameleon hash value of its children, i.e., the element $A_{i,j}$ is computed as $A_{i,j} \leftarrow \mathsf{Ch}(A_{i+1,s}, B_{i+1,s}, C_{i+1,s}; r_{i,j})$ for some integer s. So far, we have described an n-ary Merkle tree whose nodes are computed via a chameleon hash function.

Now we explain how to handle an exponential number of nodes without computing all of them. The basic idea is to exploit the collision property of the chameleon hash function. Instead of computing the node $A_{i,j}$ as $A_{i,j} \leftarrow \mathsf{Ch}(A_{i+1,s}, B_{i+1,s}, C_{i+1,s}; r_{i,j})$, we replace all elements with dummy elements, i.e., $A_{i,j} \leftarrow \mathsf{Ch}(x_{i,j}; r_{i,j})$. These elements are derived via a pseudo-random function F, i.e., $x_{i,j} \leftarrow \mathsf{F}(i,j)$, and can be computed on the fly. That is, to compute $A_{i,j}$, no other tree nodes are necessary. Since all elements are computed deterministically, it means that this modification results in an exponential number of nodes without any connection to each other. We re-establish this connection using the trapdoor of the chameleon hash function whenever we assert a new element.

We illustrate the assertion operation with Fig. 1. Assume that we would like to assert a statement in the context (associated with) $C_{3,6}$. To do so, we need to compute the the elements $A_{3,6}, B_{3,6}, A_{2,2}, B_{2,2}, A_{1,1}, C_{1,1}$ and the corresponding randomness for each node. This information will suffice for the verifier to reconstruct the *assertion path* from $C_{3,6}$ to the root as in an ordinary Merkle tree. To compute the aforementioned elements, we compute all dummy elements $x^A_{3,6}, x^B_{2,2}, x^C_{1,1}$ and we also derive the randomness for each node via F. Now, to assert the statement st in the context $C_{3,6}$, we compute the first collision in $C_{3,6} \leftarrow \mathsf{Ch}(x^C_{3,6}; r^C_{3,6})$. We use the trapdoor of the chameleon hash to find a matching randomness r' such that $\mathsf{Ch}(x^C_{3,6}; r^C_{3,6}) = C_{3,6} = \mathsf{Ch}(\mathsf{S}(st); r')$, where S computes a digest of the statement st. Now, to assert $(A_{3,6}, B_{3,6}, C_{3,6})$ with respect to the parent $C_{2,2}$, we need to find a second collision in $C_{2,2}$, which is computed as $C_{2,2} \leftarrow \mathsf{Ch}(x^C_{2,2}; r^C_{2,2})$. Again, we use the trapdoor to compute some randomness r'' such that $\mathsf{Ch}(x^C_{2,2}; r^C_{2,2}) = C_{2,2} = \mathsf{Ch}(h; r'')$ where $h = (A_{3,6}, B_{3,6}, C_{3,6})$. We repeat this procedure up to the root. Observe that independent of the statements asserted in the contexts $A_{3,6}$, $B_{3,6}$, and $C_{3,6}$, the value h will always be the same because the first collision is always computed in the leaf. This concludes the description of the underlying asserted data structure.

Now, we will explain how to extract the secret key in the case that the sender asserts two different statements in the same context. Let us assume that the sender asserted two statements st_0, st_1 in the context associated with $C_{3,6}$.

In the simplest case, there exist two pairs $(st_0, r_0), (st_1, r_1)$ such that $\mathsf{Ch}(\mathsf{S}(st_0); r_0) = C_{3,6} = \mathsf{Ch}(\mathsf{S}(st_1); r_1)$. (This is like in the "first approach".)

In a more complicated case, we could have $\mathsf{Ch}(\mathsf{S}(st_0); r_0) = C_{3,6} \neq C'_{3,6} = \mathsf{Ch}(\mathsf{S}(st_1); r_1)$, because the attacker could have used a collision in $C_{2,2}$ to associate its rightmost child with a value $C'_{3,6} \neq C_{3,6}$. But then, this collision can be used to extract the trapdoor. Generally speaking, we will find a collision somewhere on the path from the leaf to the root. Observe that this always terminates for valid assertions because a digest of the root itself is fixed in the public key.

5.2 Construction

We present the full description of our scheme. Let ℓ and n be arbitrary positive integers defining the height of a tree and its branching factor. Let F_k be a pseudorandom function, and H be a collision-resistant hash function. Furthermore, let S and L be two hash functions modeled as random oracles. L maps arbitrary bitstrings to $\{1, \ldots, n^\ell\}$.[3] Let $\mathcal{CH} = (\mathsf{GenCh}, \mathsf{Ch}, \mathsf{Col}, \mathsf{ExtractCsk})$ be a uniform, collision-resistant, and extractable chameleon hash function. The accountable assertion scheme is defined as follows:

Key Generation: The key generation algorithm chooses a key for the pseudo-random function $k \leftarrow \{0,1\}^\lambda$, and a key pair $(cpk, csk) \leftarrow \mathsf{GenCh}(1^\lambda)$ for the chameleon hash function, Let p be an unique identifier for the position of the root node. The algorithm computes the root node as $x^1_i \leftarrow \mathsf{F}_k(p, i, 0), r^1_i \leftarrow \mathsf{F}_k(p, i, 1)$, and $y^1_i = \mathsf{Ch}(x^1_i; r^1_i)$ for

[3]Since L is only required for a low failure probability of the assertion algorithm but not for security, the size of the output space of L may or may not depend on the security parameter.

$i \in \{1, \ldots, n\}$ and sets $z = \mathsf{H}(y_1^1, \ldots, y_n^1)$. Finally, it sets $apk := (cpk, z)$, $ask := csk$, and $auxsk := k$.

Assertion: The stateful assertion algorithm maintains an initially empty set L of used leaf positions. To assert a statement st in a context ct, the algorithm verifies that $\mathsf{L}(ct) \notin L$ and fails otherwise.[4] Then, it adds $\mathsf{L}(ct)$ to L and computes the assertion path $(Y_\ell, a_\ell, Y_{\ell-1}, a_{\ell-1}, \ldots, Y_1, a_1)$ from a leaf Y_ℓ to the root Y_1. Each node $Y_j = (y_1^j, \ldots, y_n^j)$ stores n entries and the position $a_j \in \{1, \ldots, n\}$ defines the position in the node. Y_ℓ is the leaf that stores the entry with the number $\mathsf{L}(ct)$, counted across all leaves from left to right, and a_ℓ is the position of this entry within Y_ℓ. In the following, let $x_i^j := \mathsf{F}_k(p_j, i, 0)$ and $r_i^j := \mathsf{F}_k(p_j, i, 1)$, where p_j is a unique identifier of the position of the node Y_j.

Compute Y_ℓ: Assert the statement st with respect to Y_ℓ by computing ${r'}_{a_\ell}^{\ell} \leftarrow \mathsf{Col}(csk, x_{a_\ell}^\ell, r_{a_\ell}^\ell, \mathsf{S}(st))$. Observe that $\mathsf{Ch}(x_{a_\ell}^\ell; r_{a_\ell}^\ell) = y_{a_\ell}^\ell = \mathsf{Ch}(\mathsf{S}(st); {r'}_{a_\ell}^\ell)$.

Compute the remaining entries in node Y_ℓ as $y_i^\ell = \mathsf{Ch}(x_i^\ell; r_i^\ell)$ for $i \in \{1, \ldots, n\} \setminus \{a_\ell\}$. The leaf Y_ℓ stores the entries $(y_1^\ell, \ldots, y_n^\ell)$. Let $z_\ell \leftarrow \mathsf{H}(y_1^\ell, \ldots, y_n^\ell)$ and let further $f_\ell = (y_1^\ell, \ldots, y_{a_\ell - 1}^\ell, y_{a_\ell+1}^\ell, \ldots, y_n^\ell)$.

Compute the nodes up to the root for $h = \ell - 1, \ldots, 1$:

- Assert z_{h+1} with respect to Y_h by computing ${r'}_{a_\ell}^h \leftarrow \mathsf{Col}(csk, x_{a_\ell}^h, r_{a_\ell}^h, z_{h+1})$. Observe that $\mathsf{Ch}(x_{a_\ell}^h; r_{a_\ell}^h) = y_{a_\ell}^h = \mathsf{Ch}(z_{h+1}; {r'}_{a_\ell}^h)$.
- Compute the remaining entries in this node Y_h as $y_i^h = \mathsf{Ch}(x_i^p; r_i^p)$ for $i \in \{1, \ldots, n\} \setminus \{a_\ell\}$. The node Y_h stores the elements $(y_1^h, \ldots, y_n^h)$. Let $z^h \leftarrow \mathsf{H}(y_1^h, \ldots, y_n^h)$ and $f_h = (y_1^h, \ldots, y_{a_\ell-1}^h, y_{a_\ell+1}^h, \ldots, y_n^h)$.

The assertion is $\tau := ({r'}_{a_\ell}^\ell, f_\ell, a_\ell, \ldots, {r'}_1^1, f_1, a_1)$.

Verification: The verification algorithm verifies that cpk is a valid chameleon hash public key and outputs 0 otherwise. It parses τ as $({r'}_{a_\ell}^\ell, f_\ell, a_\ell, \ldots, {r'}_1^1, f_1, a_1)$, and checks the validity of a statement st in a context ct by reconstructing the nodes $(Y_\ell, Y_{\ell-1}, \ldots, Y_1)$ in a bottom-up order, from the leaf Y_ℓ to the root Y_1. The verification algorithm outputs 1 if and only if $\mathsf{H}(y_1^1, \ldots, y_n^1) = z$.

Extraction: The extraction algorithm takes as input $(apk, ct, st_0, st_1, \tau_0, \tau_1)$. It computes like the verification algorithm the assertion paths for both st_0 and st_1 from the bottom up to the root until a position in the tree is found where the two assertion paths form a collision in the chameleon hash function, i.e., a position in the tree where values x_0, r_0 are used in the assertion path of st_0 and values x_1, r_1 are used in the assertion path of st_1 such that $\mathsf{Ch}(x_0; r_0) = \mathsf{Ch}(x_1; r_1)$. Then the extraction algorithm outputs the secret key $ask = csk \leftarrow \mathsf{ExtractCsk}(x_0, r_0, x_1, r_1)$ computed via the extraction algorithm of the chameleon hash function. If no such position is found, the extraction algorithm fails.

5.3 Analysis

We establish the security of the construction.

THEOREM 1. *The construction is extractable.*

PROOF. Assume for contradiction that there is a ppt attacker $\mathcal{A}$ that breaks extractability. That is, with non-negligible probability, $\mathcal{A}$ outputs a public key apk and two assertions τ_0, τ_1 that are valid for different statements $st_0 \neq st_1$ in the same context ct, but the extraction algorithm fails to extract the secret key ask given these values.

By construction of the verification algorithm, the assertion paths of τ_0 and τ_1 belong to two Merkle trees T_0 and T_1 such that *i)* the roots of T_0 and T_1 are identical, and *ii)* the two leaves of T_0 and T_1 that belong to the context ct have different inputs $st_0 \neq st_1$ for the chameleon hash function; note that these leaves are at the same position in T_0 and T_1. Thus there is a node position on the assertion paths output by $\mathcal{A}$ such that the nodes of T_0 and T_1 at this position form a collision in either the chameleon hash function or the collision-resistant hash function H. By construction of the extraction algorithm, it would not fail to output ask if this collision was a collision in the chameleon hash function. Consequently, it is a collision in the hash function H, and the existence of $\mathcal{A}$ contradicts the collision-resistance of H. □

THEOREM 2. *The construction is secret in the random oracle model.*

PROOF SKETCH. The proof proceeds in two steps. First, we prove a non-adaptive (or selective) secrecy notion where the attacker outputs all queries to the assertion oracle in the beginning. Only afterwards, the attacker obtains the public key and the responses from the oracle, and the attacker's goal is to output the secret key.

In this non-adaptive case, the reduction can answer all assertion queries without knowing the trapdoor by computing the tree from the bottom up. Whenever the attacker wins, the reduction can easily break the collision-resistance of the chameleon hash function.

In the second step, we reduce adaptive security to non-adaptive security. The reduction first outputs randomly chosen statements in randomly chosen contexts as its non-adaptive queries, and obtains a public key and the resulting assertions. Whenever the attacker queries the assertion oracle, the reduction programs the random oracles such that one of the non-adaptively obtained assertions is a valid response of the assertion oracle.

A complete proof appears in the full version [34]. □

Failure Probability of the Assertion Algorithm. The construction allows a context space of $\{0,1\}^*$. The probability that the assertion algorithm fails when given q queries is the probability that there are two contexts $ct_0 \neq ct_1$ in the queries with $\mathsf{L}(ct_0) = \mathsf{L}(ct_1)$. Under the assumption that $\mathsf{L} : \{0,1\}^* \to \{1, \ldots, n^\ell\}$ has uniform outputs, its (birthday) collision probability is below $(q+1)^2/(2 \cdot (n^\ell + 1 - q))$ [37].

Complete Accountable Assertions. A variant of the construction yields a complete accountable assertion scheme. Suppose n^ℓ is large enough (with respect to the security parameter λ) to ensure that L is collision-resistant, e.g., $\ell = 2\lambda$. Then, we can drop the check for $\mathsf{L}(ct) \notin L$, which fails now only with negligible probability. This eliminates the state from the authentication algorithm and makes the scheme complete, i.e., the assertion algorithm always succeeds.

5.4 Instantiation and Implementation

We have implemented the construction given in the previous section. In this section, we describe the details of the implementation, and we evaluate the practicality of the construction, as it will dominate the computation as well as communication costs of non-equivocation contracts. Our

[4]The set L can be implemented efficiently by a Bloom filter [9, 46], at the cost of a slightly increased failure rate. A Bloom filter is a space-efficient probabilistic data structure. It may indicate $x \in L$ wrongly with small probability but it never indicates $x \notin L$ wrongly.

implementation is available online [28]. It makes use of the libsecp256k1 library [49], which has evolved from the standard Bitcoin client.

Chameleon Hash Function. We use a chameleon hash function proposed by Krawczyk and Rabin [30], which is secure if the discrete logarithms assumption holds in the underlying group. In the elliptic curve setting, the chameleon hash function $\mathcal{CH} = (\mathsf{GenCh}, \mathsf{Ch}, \mathsf{Col})$ with extraction algorithm $\mathsf{ExtractCsk}$ is defined as follows.

$\mathsf{GenCh}(1^\lambda)$**:** The key generation algorithm chooses a secure elliptic curve and a base point g of prime order q where q is at least 2λ bits long. It chooses a random integer $\alpha \in \mathbb{Z}_q^*$ and returns $(csk, cpk) = (\alpha, X)$ with $X = g^\alpha$.

$\mathsf{Ch}(x; r)$**:** The input of the hash algorithm is a public key $cpk = X$ and a message $x \in \mathbb{Z}_q^*$. It picks a random value $r \in \mathbb{Z}_q^*$ and outputs $g^x X^r$.

$\mathsf{Col}(csk, x_0, r_0, x_1)$**:** The collision finding algorithm returns $r_1 = \alpha^{-1}(x_0 - x_1) + r_0 \pmod q$.

$\mathsf{ExtractCsk}(cpk, x_0, r_0, x_1, x_1)$**:** If the inputs are a collision, we have $g^{x_0+\alpha r_0} = g^{x_1+\alpha r_1}$. The extraction algorithm returns $\alpha = (x_0 - x_1)/(r_1 - r_0) \pmod q$.

This chameleon hash function has unique keys. A public key can be validated by verifying that it is an elliptic curve point in the correct-prime order group. To be compatible with Bitcoin keys, we work on the prime-order elliptic curve `secp256k1` [15] at a security level of 128 bits.

Cryptographic Algorithms. We use `HMAC-SHA256` to instantiate the pseudorandom function F, `SHA256` to instantiate the collision-resistant hash function H, and `HMAC-SHA256` with fixed keys to instantiate the random oracles L and S.

Other Parameters. We have chosen $\ell = 64$ as the height and $n = 2$ as the branching factor of the tree, implying that the failure probability of the assertion algorithm is below 2^{-37} for $q = 10000$ queries.

Computation Cost. On a 2.10GHz (Intel Core i7-4600U) machine with DDR3-1600 RAM, a chameleon hash evaluation takes 66 µs with a secret key, and the computation time increases to 85 µs if only a public key is available.

Let ℓ denote the height of the authentication tree. The assertion algorithm of our accountable assertion scheme in Section 5.2 requires $n\ell$ chameleon hash function evaluations using secret keys, while the verification algorithm of our accountable assertion scheme scheme requires ℓ chameleon hash function evaluations using public keys.

In our test environment, the assertion algorithm takes around 9 ms, while the verification algorithm takes approximately 4 ms to complete.

Storage Costs. A chameleon hash value is a point on the elliptic curve `secp256k1` and thus requires 257 bits < 33 bytes in compressed form. A randomness input of the chameleon hash function is an integer in the underlying field of the curve, and requires 32 bytes. An assertion is a sequence of $\ell = 64$ chameleon hash values and chameleon hash randomness inputs, and thus requires $64 \cdot (33\,\text{bytes} + 32\,\text{bytes}) = 4160$ bytes. To store $q = 10000$ assertions, we need about 42 MB.

6. NON-EQUIVOCATION CONTRACTS

Putting everything together, we explain how to realize non-equivocation contracts by combining accountable assertions and deposits. Non-equivocation contracts make it possible to penalize paltering in distributed protocols monetarily.

Setup. Let A be a party to be penalized by the loss of ₿ p if it equivocates before time T and let d be a parameter that depends on p (we will discuss the choice of d in Section 6.1).

1. Party A creates a Bitcoin key pair (pk, sk). Also, A sets up the accountable assertion scheme given in Section 5.2 with the Bitcoin key pair (pk, sk). That is, A predefines the secret key $ask := sk$ of the accountable assertion scheme and creates the corresponding public key apk and the auxiliary secret information $auxsk$ as specified in the key generation algorithm. Note that $apk = (pk, z)$ for some root hash z.
2. A creates a deposit of ₿ d with expiry time T (see Section 3.1) using pk. The deposit may or may not specify an explicit beneficiary P, who will receive the funds in case of equivocation.
3. Every party B expecting to receive asserted statements from A waits until the transaction that creates the deposit has been confirmed by the Bitcoin network.

Usage. The distributed protocol is augmented as follows:

1. Whenever A is supposed to send a statement st to different protocol parties in a context ct, party A additionally sends an assertion $\tau \leftarrow \mathsf{Assert}(ask, auxsk, ct, st)$.
2. Each recipient B verifies that $\mathsf{Verify}(apk, ct, st, \tau) = 1$ and that $T \leq t$ for the current time t. Party B ignores the message if any of the checks fail.
   Otherwise, B sends the record (apk, ct, st, τ) to the beneficiary P, who will store it. (If there is no explicit beneficiary, B publishes the record to the miners, who have an incentive to store it.)
3. If P (or the miners) detect an equivocation in two records (apk, ct, st_0, τ_0) and (apk, ct, st_1, τ_1), they use the corresponding assertions to extract A's secret key $sk \leftarrow \mathsf{Extract}(apk, ct, st_0, st_1, \tau_0, \tau_1)$. Using sk, party P transfers the funds in the deposit to an address fully under his control. (If there is no explicit beneficiary, the miners wait until the expiry time of the deposit is reached. Then each miner will try to create a block that includes a transaction transferring the deposit to an address under his control.)

If A does not equivocate, A will re-obtain full control over the deposit after its expiry.

Observe that an honest non-rational miner may not want to profit from a leaked secret key because the leakage could be the result of a security breach or carelessness. However, since the assertions constitute cryptographic proof of A's misbehavior, the miner can be assured that he acts honestly when claiming the deposit.

6.1 Analysis

We analyze the consequences of an equivocation by A.

With Explicit Beneficiary. If an explicit beneficiary P is specified in the deposit, then the properties of the deposit ensure that only P can spend the deposit in case of an equivocation. In particular, the safety margins as discussed in Section 3.1 ensure that the transaction created by P will have been confirmed already and thus the deposit will have been withdrawn already when its expiry will be reached. The size ₿ d of the deposit should be equal to the penalty ₿ p.

Without Explicit Beneficiary. If no explicit beneficiary is given, the analysis is more complicated, because a malicious sender A can participate in the mining process.

The goal of A is to establish the validity of a transaction tx that withdraws the funds in the deposit to an address controlled by A, even though her secret key has been published. Recall that such a transaction cannot be included in a block before the expiry of the deposit (Section 3.1). First, we explain how to choose the safety margin T^{impl}_{conf} to prevent A from *pre-mining* the transaction *tx*. First observe that, if T^{impl}_{conf} is too small (say $T^{impl}_{conf} = 0$ for simplicity), A can pursue the following strategy: Before the expiry time T, she tries to mine a block $\mathcal{B}$ that includes *tx* and builds upon the most current block $\mathcal{B}_{cur}$. If A manages to find such a block $\mathcal{B}$, she will keep her block $\mathcal{B}$ secret at first. If additionally no other miner finds another block $\mathcal{B}'$ building upon $\mathcal{B}_{cur}$, the malicious sender A will equivocate just before T. Then, by publishing $\mathcal{B}$ after time T, A will have a very high chance not to lose her deposit because the transaction *tx* in $\mathcal{B}$ will most likely prevail. However, if A does not manage to find a block $\mathcal{B}$, she will refrain from the equivocation attack.

This strategy is successful because the malicious sender avoids the risk of losing the deposit by performing the equivocation only if success is almost guaranteed. This is a variant of the so-called Finney attack [25].

However, assume that T^{impl}_{conf} is larger, e.g., $T^{impl}_{conf} = 60\,\text{min}$. Then 60 min before the expiry time of the deposit, A will need to have secretly pre-mined several sequential blocks (one of them containing *tx*) on top of the current block $\mathcal{B}_{cur}$ to perform the equivocation. Precisely, she will need to have pre-mined more blocks than she expects to be found by honest miners within the next 60 min. This is considered infeasible if A controls only the minority of the computation power in the network, which is the one of the underlying assumptions for security of the Bitcoin network.

While a safety margin T^{impl}_{conf} excludes pre-mining attacks, A can try to mine the first block $\mathcal{B}$ after time T. Even if other miners find a contradicting block $\mathcal{B}'$ (and maybe more sequential blocks), A can try to catch up with the blockchain, which may be worthwhile in the case of a large deposit.

We counter such attacks by a careful selection of the deposit size ₿ d. Assume that the mining power of the whole network and the A's fraction f of it stay constant. If $f < 0.5$, the probability that her block $\mathcal{B}$ prevails is $f/(1-f)$ [41]. Thus the expected penalty E for A is $E = d - d \cdot f/(1-f)$. At minimum, we require $E \geq p$, which yields $d \geq p(f-1)/(2f-1)$. For example, a deposit of $d \geq 3p/2$ is required for a malicious fraction of $f = 0.25$.

6.2 Application Examples

Many systems require users to trust in a service provider for data integrity. However, the service provider may choose to equivocate and show different users different states of the system. For instance, this has indeed been reported in the case of online social networks. A user of the Chinese microblogging service Sina Weibo claims that Sina Weibo censored his posts by not showing them to other users [44]. However, the server showed the posts to the user himself to avoid complaints from him.

To detect misbehavior of the service provider, a variety of systems have been proposed for different scenarios, e.g., SUNDR [35] for cloud storage, SPORC [24] for group collaboration, Application Transparency [22] for software distribution, and Frientegrity [23] for social networks.

They basically ensure the following property: If the server violates the *linearity* of the system by showing contradicting states to different users, it cannot merge these states again without being detected. Furthermore, if users that have received contradicting states exchange messages out of band, they can detect and prove the wrongdoing of the server. (The basic property is called *fork consistency* [12, 35]).

Observe that a violation of linearity is a case of equivocation. Although clients can cryptographically verify the append-only property, i.e., that a new system state is a proper extension of an old known system state, a malicious server can still provide different extensions to different clients.

Non-equivocation contracts are applicable in these settings. The context is a revision number of the state, and the statement is a digest of the state itself at this revision number. Depending on the system, the context may be more complex than an increasing counter. Frientegrity [23], for instance, does not maintain a total order on all operations in the system, but rather a total order per object. This is to avoid sacrificing performance. So the context in which a state is asserted is not just an increasing counter but rather the pair of (*objectID*, *perObjectRevisionNumber*).

As a concrete application, imagine a non-equivocation contract between a cloud storage provider and a client company, which is willing to pay a slightly higher usage fee as an insurance against accidental or malicious equivocation. The client company is specified as the beneficiary of the deposit. Then, the resulting contract serves as cryptographically-enforced insurance. If the service provider equivocates to individual employees of the company, the company receives the deposit.

In another example scenario, consider a market with two main providers of app stores. Both providers put down a global deposit without explicit beneficiary. If one of the providers becomes malicious and sends different binaries of the same app (and version) to different users, then it will lose its deposit. Thus, after the expiry of the deposit, the malicious provider will have to put down a new second deposit to remain in business and competitive with the honest provider, even if the loss of reputation was small. In comparison, the honest service provider can re-use the funds to put down a second deposit after the first deposit has expired. Alternatively, the malicious provider could choose not to put down a second deposit but then the honest provider can do the same while getting the funds back.

7. ASYNCHRONOUS PAYMENTS

As explained in Section 3.2, payment channels [45, 48] allow a user A to perform many transactions to a predefined recipient B up to a predefined cumulative amount ₿ d. Once the channel is established, it is possible for A to send funds to B even when both parties are offline.

However, if the recipient B is a distributed system, i.e., B actually consists of many unsynchronized entities $B_1, \ldots, B_n$, then offline transactions are not secure. The problem is that A can double-spend the same funds to B_i and B_j, who cannot talk to each other because they are offline and thus not synchronized. When B wants to close its channel and clear the payment in the Bitcoin network, it can clear these funds only once.

We can secure offline transaction through payment channels in cases where a reasonable finite penalty for double-spending can be found.

Example: Public Transport. For an illustrative example, assume B is a company offering public transport on buses. A would like to use B's services as a passenger. Thus, A establishes a payment channel to B by sending a transaction to the Bitcoin network. Once the transaction is confirmed, the payment channel is open and A can use it to pay for several single rides when she enters one of B's buses B_i up to the limit ₿ d of the channel. It is reasonable to assume that A and B have at most sporadic Internet connectivity in this mobile setting, so the payment should be performed offline. Still, B's buses are synchronized every night.

This system is flawed: A can double-spend to B's buses. Say the current state in the channel is $b = 3$. Then A can ride two (or more) buses B_i and B_j on the same day, by presenting them proof of updating the channel to $b = 4$. The bus company will only notice at night during the synchronization that it has been defrauded by A.

Using accountable assertions, we can secure this protocol. Then B can penalize the double-spending user A when closing the channel. Here, a reasonable penalty is at least the fare for a day ticket (valid for several rides on the same day).

Basic Idea. The idea of the modified protocol is as follows: Since the points of sale B_i are offline and not synchronized, we let A keep the state of the payment channel. The state consists essentially of just the current value of the channel, and a revision number of the state. To ensure that the user cannot modify the state, it is signed by the individual points of sale B_i. However, the user can still show an old signed state and re-use it. This is exactly where we can use accountable assertions: whenever the user A would like to perform a payment through the channel and claims that the latest state has revision number k, we require her to assert the statement "I buy a ticket with serial number r" in context $ct = k$, where r is a fresh nonce created by B_i. Thus, if A reuses an old signed state, her key will be extractable.

7.1 Full Protocol

Our full protocol for asynchronous payment channels consists of three phases. It uses an unforgeable signature scheme with algorithms Sign and $\mathsf{VrfySig}$, and assumes that B and its points of sale B_i have corresponding key pairs (spk_B, ssk_B) and (ssk_{B_i}, spk_{B_i}), respectively.

Setup. To create an asynchronous payment channel from A to B with amount ₿ d, penalty ₿ p, and expiry time T, the parties execute the following steps:

1. A sets up a Bitcoin key pair (pk, sk) and accountable assertions keys $(apk, ask = sk, auxsk)$ as for non-equivocation contracts (Section 6).
2. A creates a payment channel with B with amount ₿ $(d + p)$ and expiry time T (Section 3.2).
3. After the channel is confirmed by the Bitcoin network, B provides A with a signed statement $\sigma = \mathsf{Sign}(ssk_B, state)$, where $state = (T, d, k = 0, b = 0, B)$.

Payment. Whenever A would like to pay ₿ x offline at some point of sale B_i, the parties execute the following protocol:

1. B_i creates a fresh nonce r and sends it to A.
2. A sets $b := b + x$ and $\tau \leftarrow \mathsf{Assert}(ask, auxsk, k, r)$. A creates a transaction tx updating the channel to state b, and sends $(tx, \tau, state, \sigma)$ to B_i.
3. B_i receives $(tx^*, \tau^*, state^*, \sigma^*)$, parses $state^*$ as $(T^*, d^*, k^*, b^*, B_j)$, and verifies all the following conditions:
   - $\mathsf{VrfySig}(spk_{B_j}, state^*, \sigma^*) = 1$ (valid state)
   - $\mathsf{Verify}(apk, k^*, r, \tau^*) = 1$ (valid assertion)
   - tx^* is a valid transaction that updates the state of the channel to $b^* + x$
   - $b^* + x \le d^*$ (unexhausted channel)
   - $A \notin X$ (A is not blacklisted)
   - $t < T^*$ for the current time t (unexpired deposit)

   If any of the checks fail, B_i aborts the payment. Otherwise, B_i computes a new state $state' = (T^*, d^*, k^* + 1, b^* + x, B_i)$, signs it via $\sigma' \leftarrow \mathsf{Sign}(ssk_{B_i}, state')$, and sends $(state', \sigma')$ to A. B_i records tx and τ and provides service to A.
4. A updates the variables $state$ and σ with the values received from B_i.

Synchronization. At the end of each time period, B synchronizes with each point of sale B_i:

1. B collects all transactions recorded by point of sale B_i, which can delete the transactions afterwards.
2. B verifies that there are no double-spends among all transactions collected so far. If B detects that A has double-spent, B extracts A's secret key sk and uses it to sign a transaction that spends the whole payment channel worth ₿ $(d + p)$ to an address under the control of B. B adds A to the blacklist X, and sends updates of the blacklist X to each point of sale B_i.
3. Before time T, B closes the channel (Section 3.2). B adds A to the blacklist X, and sends updates of the blacklist X to each point of sale B_i.

7.2 Analysis

Observe that A can double-spend on at most one day because she will be blacklisted afterwards.

Assume A has successfully double-spent. Since all states are different, and the state contains the value b of the payment channel, she must have shown the same signed state with some revision number k twice successfully. But then, A has sent two assertions τ_0 and τ_1 that are valid in the same context $ct = k$. Since the corresponding statements st_0 and st_1 are fresh nonces, they differ with overwhelming probability. Thus B can extract A's secret key successfully, and close the payment channel at the maximum value ₿ $(d + p)$. Since the points of sale B_i accept payments only up to ₿ b, the penalty for A in case of double-spending is at least ₿ p.

8. RELATED WORK

Trusted Hardware for Non-equivocation. One way to prevent equivocation is to relay on trusted hardware assumptions [4, 19, 20, 33]. In particular, the resilience of tasks such as reliable broadcast, Byzantine agreement, and multiparty computation have been improved using a non-equivocation functionality based on a *trusted hardware module*, such as a trusted, increment-only local counter and a signature oracle, at each party.

Unlike our approach, which disincentives parties from equivocation, these systems fully prevent it, but at the same time they rely on a much stronger hardware assumption.

Smart Contracts. Crypto-currencies with more expressive, e.g., Turing-complete, script languages [11, 29] offer a simpler way to achieve non-equivocation contracts. In such systems, it is possible to create a deposit that can be opened when presented with cryptographic evidence of equivocation. As digital signatures suffice to provide such evidence and extractability is not required, they can be used instead of accountable assertions. The monetary penalty is enforced by the consensus rules of the currency. While crypto-currencies with Turing-complete languages are a very promising direction, they have not yet withstood the test of time, and their powerful languages might lead to unforeseen security issues.

Traditional E-cash. Similar to accountable assertions, Chaumian e-cash systems and one-show anonymous credential systems [5, 13, 14, 16] allow a secret to be revealed in case of double-spending. In these settings, the revealed secret is not used as a key but as the identity of the double-spender, i.e., her anonymity is revoked upon double-spending.

However, these protocols are not applicable to our scenario, because they work in a fundamentally different setting: They rely on the property that a central authority (a bank) issues coins by generating cryptographic tokens. In the decentralized Bitcoin setting, no central bank exists and cryptographic secrets are generated by the users.

9. CONCLUSION

In this paper, we introduced non-equivocation contracts in Bitcoin to penalize paltering in distributed systems.

In the process of designing these contracts, we presented a novel cryptographic primitive called accountable assertions, which reveals a predefined secret key in case of equivocation. We analyzed the security as well as the performance of our accountable assertions construction and found it to be practical for real-life use.

To prevent double-spending at unsynchronized points of sale, we further applied non-equivocation contracts to the Bitcoin network itself.

Acknowledgments

We thank Dario Fiore for insightful discussions on chameleon hash functions and the anonymous reviewers for their helpful suggestions and comments.

This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and the German Universities Excellence Initiative. Dominique Schröder is also supported by an Intel Early Career Faculty Honor Program Award.

References

[1] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. How to deal with malleability of BitCoin transactions, 2013. arXiv: *1312.3230* [CoRR].

[2] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on Bitcoin. S&P'14. IEEE.

[3] G. Ateniese and B. d. Medeiros. On the key exposure problem in chameleon hashes. SCN'04. Springer.

[4] M. Backes, F. Bendun, A. Choudhury, and A. Kate. Asynchronous MPC with a strict honest majority using non-equivocation. PODC'14. ACM.

[5] F. Baldimtsi and A. Lysyanskaya. Anonymous credentials light. CCS'13. ACM.

[6] I. Bentov and R. Kumaresan. How to use Bitcoin to design fair protocols. CRYPTO'14. Springer.

[7] Bitcoin Project. Bitcoin developer guide. *https://bitcoin.org/en/developer-guide*.

[8] Block timestamp. Entry in Bitcoin Wiki. *https://en.bitcoin.it/w/index.php?title=Block_timestamp&oldid=51392*.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[10] J. Bonneau et al. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. S&P'15. IEEE.

[11] V. Buterin. A next-generation smart contract and decentralized application platform. *https://github.com/ethereum/wiki/wiki/White-Paper*.

[12] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. PODC'07. ACM.

[13] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. EUROCRYPT'05. Springer.

[14] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. EUROCRYPT'01. Springer.

[15] Certicom. SEC 2: Recommended elliptic curve domain parameters. *http://www.secg.org/sec2-v2.pdf*.

[16] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. CRYPTO'88. Springer.

[17] CHECKLOCKTIMEVERIFY (BIP65) IsSuperMajority() soft-fork. Pull request for Bitcoin client. *https://github.com/bitcoin/bitcoin/pull/6351*.

[18] X. Chen, F. Zhang, and K. Kim. Chameleon hashing without key exposure. ISC'04. Springer.

[19] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. SOSP'07. ACM.

[20] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues. On the (limited) power of non-equivocation. PODC'12. ACM.

[21] C. Decker and R. Wattenhofer. Information propagation in the Bitcoin network. P2P'13. IEEE.

[22] S. Fahl et al. Hey, NSA: Stay away from my market! Future proofing app markets against powerful attackers. CCS '14. ACM.

[23] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with Frientegrity: privacy and integrity with an untrusted provider. USENIX Security'12. USENIX.

[24] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. OSDI'10. USENIX.

[25] H. Finney. Re: Best practice for fast transaction acceptance - how high is the risk? Post on Bitcoin forum. *https://bitcointalk.org/index.php?topic=3441.msg48384#msg48384*.

[26] M. Fitzi and U. M. Maurer. From partial consistency to global broadcast. STOC'00. ACM.

[27] C. Ho, R. v. Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate byzantine failures. NSDI'08. USENIX.

[28] Implementation of accountable assertion scheme. *http://crypsys.mmci.uni-saarland.de/projects/PenalizingEquivocation/*.

[29] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. IACR: *2015/675*.

[30] H. Krawczyk and T. Rabin. Chameleon signatures. NDSS'00. The Internet Society.

[31] J. Krupp et al. Nearly optimal verifiable data streaming (full version). 2015. IACR: *2015/333*.

[32] R. Kumaresan and I. Bentov. How to use Bitcoin to incentivize correct computations. CCS'14. ACM.

[33] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. NSDI'09. USENIX.

[34] Liar, liar, coins on fire! — Penalizing equivocation by loss of bitcoins. Full version of this paper. Additionally available in IACR ePrint Archive. 2015. *http://crypsys.mmci.uni-saarland.de/projects/PenalizingEquivocation/penalizing.pdf*.

[35] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. PODC'02. ACM.

[36] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. *https://bitcoin.org/bitcoin.pdf*.

[37] M. Peyravian, A. Roginsky, and A. Kshemkalyani. On probabilities of hash value matches. *Comput. secur.*, 17(2), 1998.

[38] B. Poettering and D. Stebila. Double-authentication-preventing signatures. ESORICS'14. Springer.

[39] J. Poon and T. Dryja. The Bitcoin Lightning Network: Scalable off-chain instant payments. Technical Report (draft). *https://lightning.network/*.

[40] Providing a deposit. Entry in Bitcoin Wiki. *https://en.bitcoin.it/w/index.php?title=Contracts&oldid=50633#Example_1:_Providing_a_deposit*.

[41] M. Rosenfeld. Analysis of hashrate-based double spending, 2014. arXiv: *1402.2009* [CoRR].

[42] D. Schröder and H. Schröder. Verifiable data streaming. CCS'12. ACM.

[43] D. Schröder and M. Simkin. VeriStream - A framework for verifiable data streaming. FC'15.

[44] S. Song. Why I left Sina Weibo. 2011. *http://songshinan.blog.caixin.com/archives/22322*.

[45] J. Spilmann. Re: Anti DoS for tx replacement. Bitcoin development mailing list. *https://www.mail-archive.com/bitcoin-development@lists.sourceforge.net/msg02028.html*.

[46] S. Tarkoma, C. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Commun. surveys and tutorials*, 14(1), 2012.

[47] P. Todd. Near-zero fee transactions with hub-and-spoke micropayments. Bitcoin development mailing list. *https://www.mail-archive.com/bitcoin-development@lists.sourceforge.net/msg06576.html*.

[48] P. Todd. OP_CHECKLOCKTIMEVERIFY. Draft for Bitcoin Improvement Proposal. *https://github.com/petertodd/bips/blob/checklocktimeverify/bip-checklocktimeverify.mediawiki*.

[49] P. Wuille et al. libsecp256k1: Optimized C library for EC operations on curve secp256k1. *https://github.com/bitcoin/secp256k1*.

APPENDIX

A. COMPARISON TO DAPS

Like accountable assertions, double-authentication-preventing signatures (DAPS) [38] prevent the authentication of different statements in the same context by providing an algorithm that extracts the secret key in case of such double-authentication.[5] DAPS are a stronger primitive than accountable assertions, with two main differences. First, there is no "auxiliary secret information" in the strongest security notion, i.e., the full secret key must be extractable in case of double-authentication. Second, DAPS are unforgeable.

Theorem 3, whose proof appears in the full version [34], captures that certain accountable assertions are DAPS.

THEOREM 3. *A secret and extractable accountable assertion scheme that is additionally complete, has a stateless assertion algorithm, and has no auxiliary secret information is a* double-signature extractable *DAPS scheme.*

It was left as an open problem to construct DAPS based on Merkle tree or chameleon hash functions [38]. We can solve these problems in the random oracle model. We modify the complete variant of the construction (Section 5.3) as follows. Instead of choosing a key k for the pseudorandom function F at random, we set $k := \mathsf{KDF}(csk)$ for a key derivation function KDF modeled as random oracle, where csk is the trapdoor of the chameleon hash function. This eliminates the auxiliary secret information. However, this modified construction achieves only extractability with trusted setup, i.e., if the key is generated honestly. (We share this limitation with the basic construction proposed by Poettering and Stebila [38].) Indeed, only the extractability of csk can be guaranteed. Suppose the attacker can generate the keys. If the attacker just choose k uniformly at random, knowing csk does not help to obtain k. Consequently, signing messages is not possible with csk alone.

Nevertheless, our modified construction is extractable with trusted setup, and it is more efficient than the construction by Poettering and Stebila [38]. On a 2.10GHz (Intel Core i7-4600U) machine with DDR3-1600 RAM, their construction takes about 6700 ms for signing and 1500 ms with asymmetric key size 2048 bits and hash size 160 bits. Our construction with corresponding parameters (in particular $\ell = 160$) takes about 23 ms for signing and 11 ms for verification. Signatures in their construction need about 40 kB, while signatures in our construction need about 4 kB.

In terms of security, our construction and their construction are only extractable with trusted setup [38]. Their construction can be made secure against malicious key generation at the cost of adding rather expensive zero-knowledge proofs to show that the public key is a well-formed Blum integer (a product of two primes p, q with $p \equiv q \equiv 3 \mod 4$). In contrast, we are not aware of any practical approach to make our construction secure without trusted setup.

A detailed discussion on the relationship between accountable assertions and DAPS appears in the full version [34]

[5]The terminology in [38] is different. While we speak of "*asserting* a statement *st* in a context *ct*", Poettering and Stebila [38] speak of "*signing* a message *st* for a subject *ct*."