Full version of an extended abstract published in ACM CCS 2016.

Transparency Overlays and Applications

Melissa Chase
Microsoft Research Redmond
melissac@microsoft.com

Sarah Meiklejohn
University College London
s.meiklejohn@ucl.ac.uk

Abstract

In this paper, we initiate a formal study of transparency, which in recent years has become an increasingly critical requirement for the systems in which people place trust. We present the abstract concept of a transparency overlay, which can be used in conjunction with any system to give it provable transparency guarantees, and then apply the overlay to two settings: Certificate Transparency and Bitcoin. In the latter setting, we show that the usage of our transparency overlay eliminates the need to engage in mining and allows users to store a single small value rather than the entire blockchain. Our transparency overlay is generically constructed from a signature scheme and a new primitive we call a dynamic list commitment, which in practice can be instantiated using a collision-resistant hash function.

1 Introduction

In the past decade, the trust that society places in centralized mechanisms run by government, network operators, and financial institutions has been eroding, with various incidents demonstrating that high integrity cannot be achieved solely through trust in one or a handful of parties. As a reaction to this erosion in trust, two alternative architectures have emerged: users have either taken matters into their own hands and flocked to systems that have no central point of trust, or they have increased pressure on central entities to provide more openness and accountability.

A prominent example of a system with no central point of trust is Bitcoin [28], which was deployed in January 2009. Bitcoin is a monetary system that is not backed by any government and is managed through a consensus mechanism over a peer-to-peer network; there is thus no single entity that issues bitcoins or validates individual transactions, and users of Bitcoin operate using pseudonyms that are not inherently tied to their real-world identity. Bitcoin has achieved staggering success: as of this writing, its market capitalization is over 8 billion USD and its underlying structure has inspired hundreds of alternative cryptocurrencies; payment gateways such as Bitpay and Coinbase allow thousands of vendors to accept it; a number of governments have taken steps to legitimize Bitcoin via interfaces with traditional financial and regulatory infrastructures; and major financial institutions such as JPMorgan Chase [30] and Nasdaq [29] have announced plans to develop Bitcoin-based technologies.

Bitcoin and its variants have achieved a large degree of success, but denying all forms of central authority arguably limits their ability to achieve widespread adoption. Thus, technological solutions have emerged that instead seek to provide more visibility into currently centralized systems. One key example of such a system is Certificate Transparency (CT) [21], which addresses shortcomings with SSL certificate authorities (CAs) — which have ranged from failing to verify the identity of even major website owners such as Google before issuing a cryptographic certificate [10, 19] to suffering major hacks [25] that result in hundreds of forged certificates being issued [22] — and empowers users to verify for themselves the correct functioning of a system with which they interact many times a day (e.g., any time they log in to a secure website, such as their email provider). Unlike Bitcoin's approach, CT does not substantially alter the underlying infrastructure (i.e., the issuance of a certificate is largely unchanged), but instead provides a way for anyone to monitor and audit the activities of CAs to ensure that bad certificates can be detected quickly, and misbehaving authorities identified and excluded.

While Bitcoin and Certificate Transparency provide solutions in different settings, they in fact share some common features; most notably, they rely on *transparency* as a means to achieve integrity. In Bitcoin, the ledger of transactions — called the blockchain — is completely transparent, meaning all Bitcoin transactions are globally visible. A similar property is provided in Certificate Transparency, in which a distributed set of servers each maintain a globally visible log of all the issued certificates of which they are aware.

Furthermore, both Bitcoin and CT adopt a *distributed* solution, which is essential to avoid placing trust in any single entity. Indeed, relying on one party creates (at worst) a system in which this central party has unilateral control over the information that is released, or (at best) a system with one central point of failure on which attackers could target their efforts. By using a solution that is both transparent and distributed, these systems intuitively provide some notion of *public auditability*: individual users can check for themselves that only "good" events have taken place (e.g., in the case of Bitcoin, that all bitcoins have been spent at most once) and detect misbehavior on the part of all actors within the system. Understanding the link between transparency and the types of misbehavior that can be detected across a variety of settings is one of the main motivations behind this work.

1.1 Our contributions.

Systems such as Bitcoin and CT seem to provide important transparency benefits (namely, the public auditability mentioned above), but the similarities and differences between their benefits are not well understood, and no formal analysis has demonstrated either the level of transparency that they provide or how this transparency provides the intended benefits. In this paper, we initiate such a formal study. In doing so, we seek to not only compare the different guarantees provided by these systems (although our analysis does accomplish this), but more importantly to create an abstract *transparency overlay* that may be used to provide these guarantees in a variety of applications beyond financial transactions and certificate issuance.

Before we can analyze these protocols or construct a transparency overlay, we must first consider the crucial components that make up these systems. Our first step is thus to formalize — in Section 3.2 — a primitive that we call a *dynamic list commitment* (DLC); a DLC can be thought of as a generalization of a rolling hash chain or hash tree, and serves as the foundation for our construction of a transparency overlay. After defining this underlying primitive, we then go on to present transparency overlays in Section 4; here our design is heavily inspired by the design of CT. We begin with a formal model for transparency overlays, and then go on to present an abstract transparency overlay and prove its security.

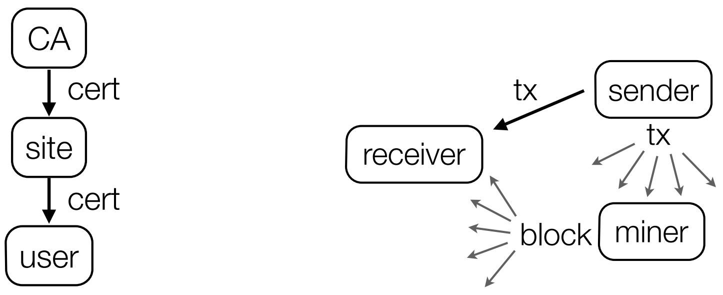
Armed with this abstract secure transparency overlay, we go on in Section 5 to demonstrate that CT is a secure transparency overlay. We also demonstrate that our formal notion of security implies more intuitive notions of security in this setting (i.e., that users should accept only "good" certificates) and discuss some practical considerations.

In Section 6, we continue by turning our attention to the Bitcoin protocol. Here, we do not use the protocol directly (as we argue that it clearly cannot satisfy our notions of security), but rather plug crucial components of the protocol into our abstract transparency overlay. While this allows us to achieve a provably secure transparency overlay for Bitcoin, it more importantly also implies that "regular" Bitcoin users (i.e., users interested only in transacting in bitcoin, rather than engaging in the mining process) can operate significantly more efficiently, provided they are willing to outsource some trust to a distributed set of parties. This result demonstrates that, in any setting in which users are willing to trust any distributed set of parties, the full decentralization of Bitcoin is not needed, and the same goals can in fact be accomplished by a CT-like structure, in which regular users store significantly less information about the transaction ledger and the mining process is superfluous; i.e., the quadrillion hashes per second expended on Bitcoin mining (as of March 2016) can be eliminated without sacrificing security. Our formal analysis thus reveals the fine line separating fully decentralized (and expensive) solutions like Bitcoin from distributed (and relatively cheap) solutions like CT, and we hope that our results can help to inform future decisions about which protocol to adopt.

1.2 Related work.

We consider research that is related both in terms of the applications of Bitcoin and Certificate Transparency, and in terms of the underlying primitives used to construct our transparency overlay.

An emerging line of work has both formalized some of the properties provided by the Bitcoin network and bootstrapped Bitcoin to obtain provably secure guarantees in other settings. Garay et al. [17] analyzed the so-called "backbone" protocol of Bitcoin and prove that it satisfies two important properties as long as the adversary controls some non-majority percentage of the hashing power. Similarly, Bentov and Kumaresan [8] provided a two-party computation built on top of (an abstracted version of) Bitcoin that provably achieves a notion of fairness, and Andrychowicz et al. [3] used Bitcoin to build a provably fair system for multi-party computation. Andrychowicz and Dziembowski [2] further formalized some of the fairness properties they require from Bitcoin (and more generally from systems based on proof-of-work) and used them to construct a broadcast protocol. Finally, on the privacy side, the Zerocash project [6] provides a cryptocurrency that has provable anonymity guarantees, and Garman et al. [18] showed how to adapt the decentralized approach of Bitcoin to achieve anonymous credentials. To the best of our knowledge, ours is the first paper to focus on the transparency property of Bitcoin, rather than its privacy or fairness guarantees.



(a) Certificate issuance. (b) Bitcoin.

Figure 1: The basic structure for each of the settings in which we apply transparency.

Aside from CT, a number of other solutions exist for changing the way we interact with certificate authorities; many of these solutions require a ground-up redesign of the CA ecosystem, which is why we chose to examine CT instead and use it as our inspiration for an overlay system. Fromknecht et al. [16] propose a decentralized PKI, based on Bitcoin and Namecoin, that eliminates the trust in centralized authorities altogether. CONIKS [24] provides an approach to logging certificates that differs from CT in two key ways: it focuses on user rather than website certificates, and largely because of this it provides a privacy-preserving solution, in which certain aspects of the stored certificates (e.g., usernames) are kept hidden. The Accountable Key Infrastructure [20] and the related ARPKI [4] both require a distributed infrastructure for not only the storage of issued certificates (as CT does), but also for their issuance, thus focusing on the prevention rather than just detection of misbehavior. Ryan [33] demonstrated how to extend CT to handle revocation of certificates. In a concurrent work, Dowling et al. [15] provided a different security model for CT and demonstrated that if various properties of the underlying Merkle trees are satisfied then CT is provably secure in their model. Although somewhat overlapping with our own work, their paper is focused firmly on CT and not on the abstract properties of transparency overlays and how they can be applied across a variety of settings.

Finally, the main primitive underlying our transparency overlay (a dynamic list commitment) is primarily a generalization of a Merkle tree [26], and is similar to the definition of a tamper-evident log given by Crosby and Wallach [13]. It is also related to the notion of an authenticated data structure (ADS) [1, 32, 31] and the notion of a cryptographic accumulator [7, 12, 11, 23]; indeed the application of ADSs to Bitcoin has already been touched on in previous work [27] (but from the perspective of programming languages, and thus without any consideration of security). Dynamic list commitments differ from these related primitives in terms of the security model, however, and as a result we can provide more efficient constructions while still satisfying a notion of provable security.

2 Background

2.1 Certificate Transparency

Certificate Transparency (CT) was proposed in 2011 by Ben Laurie and Adam Langley as a way to increase transparency in the process of issuing certificates, so that certificate authorities (CAs) can be held responsible for their actions and bad certificates can be caught and revoked early on. The basic process of issuing certificates operates as depicted in Figure 1a: a CA issues a certificate to a website operator, who then publishes this certificate so that users can check it. Certificate Transparency then provides an extra layer on top of this basic interaction to provide transparency; in fact, as we will see in Section 4, our design of a transparency overlay is heavily inspired by the CT design.

Briefly, CT introduces three additional actors: a log server, who is responsible for keeping track of issued certificates, an auditor, who is responsible (on behalf of the client) for keeping track of whether given certificates are in the log or not, and a monitor, who is responsible for checking the quality of the certificates in the log. As we use these additional actors in our general transparency overlay, we defer further discussion of their roles and actions to Section 4.1. In Section 5, we prove that CT provides a provably secure transparency overlay, thus (provably) providing the intuitive security properties that one would hope to achieve.

2.2 Bitcoin

Bitcoin is a decentralized cryptocurrency that was introduced in 2008 [28] and deployed on January 3 2009. We briefly sketch the main properties of Bitcoin and its underlying *blockchain* technology here, and refer the reader to Bonneau et al. [9] for a more comprehensive overview.

Briefly, Bitcoin operates as depicted in Figure 1b. A sender, identified using a pseudonym or *address*, has some number of bitcoins stored with this address; i.e., within the Bitcoin network, this address is acknowledged as the owner of these bitcoins. To transfer ownership of these bitcoins to some receiver, the sender first creates a transaction to send them to the receiver, as identified by whichever address she has given to the sender. The transaction is signed to ensure that only the sender can give away his own bitcoins.

After forming this transaction, the sender broadcasts it to the Bitcoin network, where it eventually reaches a miner, who acts to seal the transaction into a block. The miner broadcasts this block, containing the transaction, to the network, where it eventually reaches the receiver, who can confirm the transaction and its position within the Bitcoin ledger (i.e., the blockchain) to satisfy herself that she is now the owner of the bitcoins.

Because the Bitcoin blockchain is globally visible, it already provides a degree of transparency that is higher than that of traditional financial transactions. In Section 6, we apply a transparency overlay on top of Bitcoin and demonstrate that it provides a significantly more efficient way for Bitcoin users to participate in transactions and allows hashing to be eliminated from the system.

3 Definitions and Notation

In this section, we define various notions that will be used throughout the rest of the paper. In particular, we formalize *dynamic list commitments* in Section 3.2, which can be thought of as a generalization of Merkle trees and allow us to construct high-integrity logs.

3.1 Preliminaries

If x is a binary string then $|x|$ denotes its bit length. If S is a finite set then $|S|$ denotes its size and $x \xleftarrow{r} S$ denotes sampling a member uniformly from S and assigning it to x. $\lambda \in \mathbb{N}$ denotes the security parameter and 1^λ denotes its unary representation. ε denotes the null value.

Algorithms are randomized unless explicitly noted otherwise. "PT" stands for "polynomial-time." By $y \leftarrow A(x_1, \ldots, x_n; R)$ we denote running algorithm A on inputs $x_1, \ldots, x_n$ and random coins R and assigning its output to y. By $y \xleftarrow{r} A(x_1, \ldots, x_n)$ we denote $y \leftarrow A(x_1, \ldots, x_n; R)$ for coins R sampled uniformly at random. By $[A(x_1, \ldots, x_n)]$ we denote the set of values that have positive probability of being output by A on inputs $x_1, \ldots, x_n$. Adversaries are algorithms.

For interactive protocols, we use the notation of Bellare and Keelveedhi [5]. For completeness, we include the formal notion of defining and executing interactive protocols in Appendix A. Briefly, the behavior of a stateful participant party that is given m during the i-th round of the j-th execution of a protocol Prot can be defined as $(\mathsf{state}_{\mathsf{party}}, m', p, \mathsf{out}) \xleftarrow{r} \mathsf{Prot}[\mathsf{party}, i, j](1^\lambda, \mathsf{state}_{\mathsf{party}}, m)$, where p indicates the party to which it is sending m'; the execution of the entire interactive protocol can be defined by $\mathsf{outputs} \xleftarrow{r} \mathsf{Run}(1^\lambda, \mathsf{Prot}, \mathsf{Parties}, \mathsf{inputs})$; and the message sent during the protocol (i.e., the transcript) can be defined by $M \xleftarrow{r} \mathsf{Msgs}(1^\lambda, \mathsf{Prot}, \mathsf{Parties}, \mathsf{inputs})$.

We use games in security definitions and proofs. A game G has a MAIN procedure whose output is the output of the game. $\Pr[\mathsf{G}]$ denotes the probability that this output is true.

3.2 Dynamic list commitments

We define a *dynamic list commitment* (DLC), which allows one to commit to a list of elements in such a way that (1) the list represented by the commitment can be updated only by having new elements appended to the end, and (2) given just the list commitment, one can efficiently prove both the append-only property of the list and that a given element is in the list.

One common example of a DLC is a hash tree, and in particular a Merkle tree, in which the root hash acts as the commitment and one can use the hashes of intermediate nodes to prove the above properties. (Indeed, this is what CT uses.) Our basic formalization is similar to the definition of a tamper-evident history system [13], but we also include an augmented version that considers additional properties one can use when operating on *ordered* lists.

3.2.1 A basic formalization for general lists.

We define a dynamic list commitment DLC as a collection of the following algorithms:

- $\boldsymbol{c} \leftarrow \mathsf{\textbf{Com(list)}}$ creates the commitment c and $\mathbf{0/1} \leftarrow \mathsf{\textbf{CheckCom}}(\boldsymbol{c}, \mathsf{\textbf{list}})$ checks that c is a commitment to list;
- $\boldsymbol{c}_{\mathsf{new}} \leftarrow \mathsf{\textbf{Append}}(\mathsf{\textbf{list}}_{\Delta}, \boldsymbol{c}_{\mathsf{old}})$ updates the commitment to take into account the new elements in list_{Δ};
- $\boldsymbol{\pi} \leftarrow \mathsf{\textbf{ProveAppend}}(\boldsymbol{c}_{\mathsf{old}}, \boldsymbol{c}_{\mathsf{new}}, \mathsf{\textbf{list}})$ proves that c_{new} was obtained from c_{old} solely by appending elements to an earlier version of list and $\mathbf{0/1} \leftarrow \mathsf{\textbf{CheckAppend}}(\boldsymbol{c}_{\mathsf{old}}, \boldsymbol{c}_{\mathsf{new}}, \boldsymbol{\pi})$ checks this proof;
- $\boldsymbol{\pi} \leftarrow \mathsf{\textbf{ProveIncl}}(\boldsymbol{c}, \mathsf{\textbf{elmt}}, \mathsf{\textbf{list}})$ proves that elmt is in list (as represented by c); and $\mathbf{0/1} \leftarrow \mathsf{\textbf{CheckIncl}}(\boldsymbol{c}, \mathsf{\textbf{elmt}}, \boldsymbol{\pi})$ checks this proof.

Definition 3.1. *A DLC is* correct *if the following properties are satisfied for all lists* list *and* list_{Δ}*, for all* $\mathsf{elmt} \in \mathsf{list}$*, and for* $c \leftarrow \mathsf{Com}(\mathsf{list})$*:*

- $\mathsf{CheckCom}(c, \mathsf{list}) = 1$*;*
- $\mathsf{CheckIncl}(c, \mathsf{elmt}, \mathsf{ProveIncl}(c, \mathsf{elmt}, \mathsf{list})) = 1$*;*
- $\mathsf{Append}(\mathsf{list}_{\Delta}, c) = \mathsf{Com}(\mathsf{list} \| \mathsf{list}_{\Delta})$*; and*
- $\mathsf{CheckAppend}(c, \mathsf{Append}(\mathsf{list}_{\Delta}, c), \mathsf{ProveAppend}(c, \mathsf{Append}(\mathsf{list}_{\Delta}, c), \mathsf{list})) = 1$.

We say that a DLC is *compact* if $\mathsf{Com}(\mathsf{list})| \ll |\mathsf{list}|$ for all sufficiently long lists list. For security, intuitively a DLC should be (1) *binding*, which means that a commitment cannot represent two different lists, (2) *sound*, which means that it should be hard to produce a valid proof of inclusion for an element that is not in the list, and (3) *append-only*, which means that it should be hard to produce a proof that a list has only been appended to if other operations (e.g., deletions) have taken place. Formally, these properties can be defined as follows:

Definition 3.2 (Basic)**.** *Define* $\mathbf{Adv}^{s}_{dlc,\mathcal{A}}(\lambda) = Pr[\mathsf{G}^{\mathcal{A}}_{s}(\lambda)]$ *for* $s \in \{bind, sound, append\}$*, where these games are defined as follows:*

```
MAIN $\mathsf{G}^{bind}_{\mathcal{A}}(\lambda)$
$(c, \mathsf{list}_1, \mathsf{list}_2) \xleftarrow{r} \mathcal{A}(1^{\lambda})$
return $(\mathsf{CheckCom}(c, \mathsf{list}_1) \wedge \mathsf{CheckCom}(c, \mathsf{list}_2) \wedge (\mathsf{list}_1 \neq \mathsf{list}_2))$

MAIN $\mathsf{G}^{sound}_{\mathcal{A}}(\lambda)$
$(c, \mathsf{list}, \mathsf{elmt}, \pi) \xleftarrow{r} \mathcal{A}(1^{\lambda})$
return $(\mathsf{CheckCom}(c, \mathsf{list}) \wedge \mathsf{CheckIncl}(c, \mathsf{elmt}, \pi) \wedge (\mathsf{elmt} \notin \mathsf{list}))$

MAIN $\mathsf{G}^{append}_{\mathcal{A}}(\lambda)$
$(c_1, c_2, \mathsf{list}_2, \pi) \xleftarrow{r} \mathcal{A}(1^{\lambda})$
return $(\mathsf{CheckCom}(c_2, \mathsf{list}_2) \wedge \mathsf{CheckAppend}(c_1, c_2, \pi) \wedge (\nexists j : \mathsf{CheckCom}(c_1, \mathsf{list}_2[1:j])))$
```

Then the DLC is binding *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{bind}_{dlc,\mathcal{A}}(\lambda) < \nu(\lambda)$*,* sound *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{sound}_{dlc,\mathcal{A}}(\lambda) < \nu(\lambda)$*, and* append-only *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{append}_{dlc,\mathcal{A}}(\lambda) < \nu(\lambda)$*.*

3.2.2 An augmented formalization for ordered lists.

It will also be useful for us to consider a special type of DLC, in which the elements in the list have some kind of order imposed on them. In particular, this allows us to more efficiently perform two additional operations: demonstrate that two DLCs are inconsistent (i.e., that they are commitments to strictly distinct or forking lists), and demonstrate that a given element is not in the list represented by a given commitment. As we will see in our applications later on, these operations are crucial for providing evidence that certain types of misbehavior have taken place.

In addition to the algorithms required for a basic DLC, we now require a notion of timing (which may not be the actual time, but rather any representation that allows us to impose an ordering): for every element elmt in a list, we assume there exists a function $\mathsf{time}(\cdot)$ that returns a value t, and that a global ordering exists for this function, so that we can also define a Boolean function $0/1 \leftarrow \mathsf{isOrdered}(\mathsf{list})$. Using this, we define a notion of *consistency* for DLCs as follows:

Definition 3.3 (Consistency)**.** *A tuple* (c, t, list) *is* consistent *if* c *is a commitment to the state of* list *at time* t*. Formally, we consider a function* $\mathsf{isConsistent}$ *such that* $\mathsf{isConsistent}(c, t, \mathsf{list}) = 1$ *if and only if there exists a* j*,* $1 \leq j \leq \mathsf{len}(\mathsf{list})$*, such that (1)* $\mathsf{CheckCom}(c, \mathsf{list}[1:j]) = 1$*, (2)* $\mathsf{time}(\mathsf{list}[j]) \leq t$*, (3)* $j = \mathsf{len}(\mathsf{list})$ *or* $\mathsf{time}(\mathsf{list}[j+1]) \geq t$*), and (4)* $\mathsf{isOrdered}(\mathsf{list})$*.*

We can now define four additional algorithms as follows:

- $\pi \leftarrow \textsf{\textbf{DemoInconsistent}}(\textsf{list}, c', t')$ proves that $\textsf{list}$ is inconsistent with c' at time t' and
- $0/1 \leftarrow \textsf{\textbf{CheckInconsistent}}(c', t', c, \pi)$ checks this proof;
- $\pi \leftarrow \textsf{\textbf{DemoNotIncl}}(\textsf{list}, \textsf{elmt})$ proves that $\textsf{elmt}$ is not in the ordered list $\textsf{list}$; and
- $0/1 \leftarrow \textsf{\textbf{CheckNotIncl}}(c, \textsf{elmt}, \pi)$ checks this proof.

The security properties in the augmented setting are relatively intuitive: an adversary should not be able to (1) produce an inconsistent tuple $(c, t, \textsf{list})$ such that one cannot demonstrate their inconsistency, (2) produce an ordered list and element such that one cannot demonstrate the non-inclusion of the element in the list, (3) demonstrate an inconsistency that does not exist, or (4) prove non-inclusion of an element that is in fact in an ordered list. We define these formally as follows:[1]

Definition 3.4 (Augmented). *Define* $\mathbf{Adv}^{s}_{dlc,\mathcal{A}}(\lambda) = Pr[\mathsf{G}^{\mathcal{A}}_{s}(\lambda)]$ *for* $s \in \{p\text{-}cons, p\text{-}incl, uf\text{-}cons, uf\text{-}incl\}$*, where these games are defined as follows:*

```
MAIN $\mathsf{G}^{p\text{-}cons}_{\mathcal{A}}(\lambda)$
$(c, t, \textsf{list}) \xleftarrow{r} \mathcal{A}(1^\lambda)$
$b \leftarrow \textsf{CheckInconsistent}(c, t, \textsf{Com}(\textsf{list}), \textsf{DemoInconsistent}(\textsf{list}, c, t))$
return $((b = 0) \wedge (\textsf{isConsistent}(c, t, \textsf{list}) = 0) \wedge \textsf{isOrdered}(\textsf{list}))$

MAIN $\mathsf{G}^{p\text{-}incl}_{\mathcal{A}}(\lambda)$
$(\textsf{list}, \textsf{elmt}) \xleftarrow{r} \mathcal{A}(1^\lambda)$
$b \leftarrow \textsf{CheckNotIncl}(\textsf{Com}(\textsf{list}), \textsf{elmt}, \textsf{DemoNotIncl}(\textsf{list}, \textsf{elmt}))$
return $((b = 0) \wedge (\textsf{elmt} \notin \textsf{list}) \wedge \textsf{isOrdered}(\textsf{list}))$

MAIN $\mathsf{G}^{uf\text{-}cons}_{\mathcal{A}}(\lambda)$
$(c_1, t, c_2, \textsf{list}_2, \pi) \xleftarrow{r} \mathcal{A}(1^\lambda)$
return $(\textsf{CheckCom}(c_2, \textsf{list}_2) \wedge \textsf{isConsistent}(c_1, t, \textsf{list}_2) \wedge \textsf{CheckInconsistent}(c_1, t, c_2, \pi))$

MAIN $\mathsf{G}^{uf\text{-}incl}_{\mathcal{A}}(\lambda)$
$(c, \textsf{list}, \textsf{elmt}, \pi) \xleftarrow{r} \mathcal{A}(1^\lambda)$
return $(\textsf{CheckCom}(c, \textsf{list}) \wedge (\textsf{elmt} \in \textsf{list}) \wedge \textsf{CheckNotIncl}(c, \textsf{elmt}, \pi) \wedge \textsf{isOrdered}(\textsf{list}))$
```

Then the DLC satisfies provable inconsistency *if for all PT adversaries* $\mathcal{A}$ $\mathbf{Adv}^{p\text{-}cons}_{dlc,\mathcal{A}}(\lambda) = 0$*,* provable non-inclusion *if for all PT adversary* $\mathcal{A}$ $\mathbf{Adv}^{p\text{-}incl}_{dlc,\mathcal{A}}(\lambda) = 0$*,* unforgeable inconsistency *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{uf\text{-}cons}_{dlc,\mathcal{A}}(\lambda) < \nu(\lambda)$*, and* unforgeable non-inclusion *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{uf\text{-}incl}_{dlc,\mathcal{A}}(\lambda) < \nu(\lambda)$*.*

3.2.3 Two instantiations of augmented DLCs.

To demonstrate that dynamic list commitments exist, we provide two instantiations; both can be found in Appendix B and derive their security from the collision resistance of a hash function. Briefly, our first instantiation is essentially a rolling hash chain: new elements appended to the list are folded into the hash (i.e., $c_{\textsf{new}} \leftarrow H(c_{\textsf{old}} \| \textsf{elmt}_{\textsf{new}})$), and proofs about (in)consistency and (non-)inclusion reveal selective parts of the list. This first instantiation thus demonstrates the feasibility of dynamic list commitments (and is conceptually quite simple), but the proofs are linear in the size of the list, which is not particularly efficient. Thus, our second instantiation is essentially a Merkle tree, which allows us to achieve proofs that are logarithmic in the size of the list.

4 Transparency Overlays

In this section, we present our main contributions. First, in Sections 4.1 and 4.2, we introduce both basic and augmented formal models for reasoning about transparency. Then, in Sections 4.3 and 4.4 we present a generic transparency overlay and prove its security. To instantiate this securely (as we do in Sections 5 and 6), one then need only provide a simple interface between the underlying system and the overlay.

[1] In our games below, we require as a winning condition that the list is ordered. This is because our constructions make this assumption in order to achieve better efficiency, but one could also present constructions for which this extra winning condition would not be needed.

4.1 Basic overlays

In order for a system to be made transparent, we must provide an efficient mechanism for checking that the system is running correctly. Our setting overlays three additional parties on top of an existing system Sys: a *log server* LS, an *auditor* $\mathsf{Auditor}$, and a *monitor* $\mathsf{Monitor}$. The role of the log server is to take certain events in the system's operation and enter them into a publicly available log. The role of the auditor is to check — crucially, without having to keep the entire contents of the log — that specific events are in the log. Finally, the role of the monitor is to flag any problematic entries within the log. Collectively then, the auditor and monitor act to hold actors within the system responsible for the creation of (potentially conflicting) events.

We assume that each of these parties is stateful: the log server maintains the log as state, so $\mathsf{state}_{\mathsf{LS}} = \mathsf{log}$; the auditor maintains a *snapshot* (i.e., some succinct representation of the current log) as state, so $\mathsf{state}_{\mathsf{Au}} = \mathsf{snap}$; and the monitor maintains a snapshot, a list of bad events, and a list of all events, so $\mathsf{state}_{\mathsf{Mo}} = (\mathsf{snap}, \mathsf{events}_{\mathsf{bad}}, \mathsf{events})$.

A transparency overlay then requires five interactive protocols; these are defined abstractly as follows:[2]

`GenEventSet` is an interaction between the actor(s) in the system that produces the events to be logged. The protocol is such that $\mathbf{eventset} \xleftarrow{r} \mathbf{Run}(1^\lambda, \mathtt{GenEventSet}, \mathbf{Sys}, \mathbf{aux})$.

`Log` is an interaction between one or more of the actors in the system and LS that is used to enter events into the log. The protocol is such that $(b, \varepsilon) \xleftarrow{r} \mathbf{Run}(1^\lambda, \mathtt{Log}, (\mathbf{Sys}, \mathbf{LS}), (\mathbf{eventset}, \varepsilon))$, where b indicates whether or not the system actor(s) believes the log server behaved honestly.

`CheckEntry` is an interaction between one or more of the actors in the system, $\mathsf{Auditor}$, and LS that is used to check whether or not an event is in the log. The protocol is such that $(b, b', \varepsilon) \xleftarrow{r} \mathbf{Run}(1^\lambda, \mathtt{CheckEntry}, (\mathbf{Sys}, \mathbf{Auditor}, \mathbf{LS}), (\mathbf{event}, \varepsilon, \varepsilon))$, where b indicates whether or not the system actor(s) believes the event to be in the log and b' indicates whether or not the auditor believes the log server behaved honestly in the interaction.

`Inspect` is an interaction between $\mathsf{Monitor}$ and LS that is used to allow the monitor to inspect the contents of the log and flag any suspicious entries. The protocol is such that $(b, \varepsilon) \xleftarrow{r} \mathbf{Run}(1^\lambda, \mathtt{Inspect}, (\mathbf{LS}, \mathbf{Monitor}), (\varepsilon, \varepsilon))$, where b indicates whether or not the monitor believes the log server behaved honestly in the interaction.

`Gossip` is an interaction between $\mathsf{Auditor}$ and $\mathsf{Monitor}$ that is used to compare versions of the log and detect any inconsistencies. If any misbehavior on behalf of the log server is found, then both parties are able to output *evidence* that this has taken place. The protocol is such that $(\mathbf{evidence}, \mathbf{evidence}) \xleftarrow{r} \mathbf{Run}(1^\lambda, \mathtt{Gossip}, (\mathbf{Auditor}, \mathbf{Monitor}), (\varepsilon, \varepsilon))$.

We also require the following (non-interactive) algorithms:

$(pk_{\mathsf{LS}}, sk_{\mathsf{LS}}) \xleftarrow{r} \mathbf{GenLogID}(1^\lambda)$ is used to generate a public and secret identifier for the log server; and

$0/1 \leftarrow \mathbf{CheckEvidence}(pk_{\mathsf{LS}}, \mathbf{evidence})$ is used to check if the evidence against the log server identified by pk_{LS} is valid.

From a functionality standpoint, we would like the protocols to be *correct*, meaning all parties should be satisfied by honest interactions, and *compactly auditable*, meaning the size of a snapshot is much smaller than the size of the log.

We define security for a basic transparency overlay in terms of two properties: *consistency*, which says that a potentially dishonest log server cannot get away with presenting inconsistent versions of the log to the auditor and monitor, and *non-frameability*, which says that potentially dishonest auditors and monitors (and even actors in the original system) cannot blame the log server for misbehavior if it has behaved honestly. Participants can thus be satisfied that they are seeing the same view of the log as all other participants, and that the interactions they have really are with the log server.

To formalize consistency, we consider a game in which the adversary takes on the role of the log server and is allowed to interact (via the MsgAu and MsgMo oracles, respectively) with the auditor and monitor. The adversary wins if there is an event that is not in the list maintained by the monitor but that the auditor nevertheless perceives as being in the log (the third winning condition of Definition 4.1), yet the auditor and monitor are unable to produce valid evidence of this inconsistency (the first two winning conditions). For ease of formal exposition, we (1) assume that in the `CheckEntry` protocol the first message sent to the auditor is the event to be checked and the last message sent by the auditor is a bit indicating whether the event is in the log, and (2) require that the monitor must have a newer snapshot than the auditor, but can naturally extend our definition to cover other configurations as well.

[2] In each protocol, we also allow the participants to output fail, which indicates that they believe they were given improperly formatted inputs.

Definition 4.1 (Consistency). *Define* $\mathbf{Adv}^{cons}_{trans,\mathcal{A}}(\lambda) = Pr[\mathsf{G}^{cons}_{\mathcal{A}}(\lambda)]$*, where* $\mathsf{G}^{cons}_{\mathcal{A}}(\lambda)$ *is defined as follows:*

```
$\underline{\text{MAIN } \mathsf{G}^{cons}_{\mathcal{A}}(\lambda)}$
$\mathsf{events} \leftarrow \emptyset;\ \mathsf{events}_{\mathsf{pass}} \leftarrow \emptyset$
$pk_{\mathsf{LS}} \xleftarrow{r} \mathcal{A}^{\textsc{MsgAu},\textsc{MsgMo}}(1^\lambda)$
$\mathsf{evidence} \xleftarrow{r} \mathsf{Run}(1^\lambda, \mathtt{Gossip}, (\mathsf{Auditor}, \mathsf{Monitor}), (\varepsilon, \varepsilon))$
$return$ $((\mathsf{CheckEvidence}(pk_{\mathsf{LS}}, \mathsf{evidence}) = 0) \quad \wedge$
        $(\mathsf{time}(\mathsf{state}_{\mathsf{Mo}}[\mathsf{snap}]) \geq \mathsf{time}(\mathsf{state}_{\mathsf{Au}}[\mathsf{snap}])) \quad \wedge$
        $(\mathsf{events}_{\mathsf{pass}} \setminus \mathsf{state}_{\mathsf{Mo}}[\mathsf{events}] \neq \emptyset))$

$\underline{\textsc{MsgAu}(i, j, m)}$
$(\mathsf{state}_{\mathsf{Au}}, m', p, \mathsf{out}) \xleftarrow{r} \mathtt{CheckEntry}[\mathsf{Auditor}, i, j](1^\lambda, \mathsf{state}_{\mathsf{Au}}, m)$
$if$ $(i = 1)$ $\mathsf{events}[j] \leftarrow m$
$if$ $(\mathsf{out} \neq \bot) \wedge (m' = 1)$ $\mathsf{events}_{\mathsf{pass}} \leftarrow \mathsf{events}_{\mathsf{pass}} \cup \{\mathsf{events}[j]\}$
$return$ $m'$

$\underline{\textsc{MsgMo}(i, j, m)}$
$(\mathsf{state}_{\mathsf{Mo}}, m', p, \mathsf{out}) \xleftarrow{r} \mathtt{Inspect}[\mathsf{Monitor}, i, j](1^\lambda, \mathsf{state}_{\mathsf{Mo}}, m)$
$return$ $m'$
```

Then the transparency overlay satisfies consistency *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{cons}_{trans,\mathcal{A}}(\lambda) < \nu(\lambda)$.

Next, to formalize non-frameability, we consider an adversary that wants to frame an honest log server; i.e., to produce evidence of its "misbehavior." In this case, we consider a game in which the adversary takes on the role of the auditor, monitor, and any actors in the system, and is allowed to interact (via the Msg oracle) with the honest log server. The adversary wins if it is able to produce evidence that passes verification.

Definition 4.2 (Non-frameability). *Define* $\mathbf{Adv}^{frame}_{trans,\mathcal{A}}(\lambda) = Pr[\mathsf{G}^{frame}_{\mathcal{A}}(\lambda)]$*, where* $\mathsf{G}^{frame}_{\mathcal{A}}(\lambda)$ *is defined as follows:*

```
$\underline{\text{MAIN } \mathsf{G}^{frame}_{\mathcal{A}}(\lambda)}$
$(pk_{\mathsf{LS}}, sk_{\mathsf{LS}}) \xleftarrow{r} \mathsf{GenLogID}(1^\lambda)$
$\mathsf{evidence} \xleftarrow{r} \mathcal{A}^{\textsc{Msg}}(1^\lambda, pk_{\mathsf{LS}})$
$return$ $\mathsf{CheckEvidence}(pk_{\mathsf{LS}}, \mathsf{evidence})$

$\underline{\textsc{Msg}(\mathsf{Prot}, i, j, m)}$
$if$ $(\mathsf{Prot} \notin \{\mathtt{Log}, \mathtt{CheckEntry}, \mathtt{Inspect}\})$ $return$ $\bot$
$(\mathsf{state}_{\mathsf{LS}}, m', p, \mathsf{out}) \xleftarrow{r} \mathsf{Prot}[\mathsf{LS}, i, j](1^\lambda, \mathsf{state}_{\mathsf{LS}}, m)$
$return$ $m'$
```

Then the transparency overlay satisfies non-frameability *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{frame}_{trans,\mathcal{A}}(\lambda) < \nu(\lambda)$.

We then say that a basic transparency overlay is *secure* if it satisfies consistency and non-frameability.

Comparison with concurrent work With respect to the security model of Dowling et al. [15], their model requires only that the monitor and auditor produce evidence of misbehavior in the case where the log fails to include an event for which it has issued a receipt (which we consider in the next section). Our model, on the other hand, also produces evidence in the case where the log has given inconsistent views to the two parties; this type of evidence seems particularly valuable since this type of misbehavior is only detected after the fact. This difference allows them to present a simpler definition of non-frameability, as they do not have to worry about malicious monitors and auditors forging this type of evidence.

4.2 Pledged overlays

In the basic setting described, log servers can be held responsible if they attempt to present different views of the log to the auditor and monitor. If log servers simply fail to include events in the log in the first place, however, then there is currently no way to capture this type of misbehavior. While in certain settings the log server could plausibly claim that it never received an event rather than ignoring it, if the log server issues promises or *receipts* to include events in the log then we can in fact enforce inclusion, or at least blame the log server if it fails to do so.

Formally, we capture this as an additional security property, *accountability*, which says that evidence can also be used to implicate log servers that promised to include events but then did not. In the game, the adversary then takes on the role of the log server and is allowed to interact arbitrarily with the actor(s) in the system, auditor, and monitor. It wins if there is an event that it has pledged to include but that the auditor and monitor do not believe to be in the log (the third winning condition of Definition 4.3), yet the auditor and monitor are unable to produce evidence of this omission (the first two winning conditions). For ease of formal exposition, we assume Sys produces its final output before Auditor in the `CheckEntry` protocol, but again note that this does not sacrifice generality.

Definition 4.3 (Accountability). *Define* $\mathbf{Adv}^{trace}_{trans,\mathcal{A}}(\lambda) = Pr[\mathsf{G}^{trace}_{\mathcal{A}}(\lambda)]$*, where* $\mathsf{G}^{trace}_{\mathcal{A}}(\lambda)$ *is defined as follows:*

```
MAIN $\mathsf{G}^{trace}_{\mathcal{A}}(\lambda)$
$\mathsf{events}_1, \mathsf{events}_2, \mathsf{events}_{\mathsf{pledged}}, \mathsf{events}_{\mathsf{fail}}, a \leftarrow \emptyset$
$pk_{\mathsf{LS}} \xleftarrow{r} \mathcal{A}^{\textsc{MsgSys},\textsc{MsgAu},\textsc{MsgMo}}(1^\lambda)$
$\mathsf{evidence} \xleftarrow{r} \mathsf{Run}(1^\lambda, \mathtt{Gossip}, (\mathsf{Auditor}, \mathsf{Monitor}), (\varepsilon, \varepsilon))$
$return\ ((\mathsf{CheckEvidence}(pk_{\mathsf{LS}}, \mathsf{evidence}) = 0) \quad \wedge$
        $(\mathsf{time}(\mathsf{state}_{\mathsf{Mo}}[\mathsf{snap}]) \geq \mathsf{time}(\mathsf{state}_{\mathsf{Au}}[\mathsf{snap}])) \wedge$
        $((\mathsf{events}_{\mathsf{pledged}} \cap \mathsf{events}_{\mathsf{fail}}) \setminus \mathsf{state}_{\mathsf{Mo}}[\mathsf{events}] \neq \emptyset))$

$\textsc{MsgSys}(\mathsf{Prot}, i, j, m)$
$a[i,j] \leftarrow m$
$if\ (\mathsf{Prot} = \mathtt{Log})$
    $if\ (i = 1)\ \mathsf{events}_1[j] \leftarrow m[\mathsf{events}]$
    $if\ (\mathsf{out} = 1)\ \mathsf{events}_{\mathsf{pledged}} \leftarrow \mathsf{events}_{\mathsf{pledged}} \cup \mathsf{events}_1[j]$
$if\ (\mathsf{Prot} = \mathtt{CheckEntry}) \wedge (\mathsf{out} = 0)\ \mathsf{events}_{\mathsf{fail}}[j] \leftarrow \mathsf{events}_2[j]$
$(a[i+1,j], m', p, \mathsf{out}) \xleftarrow{r} \mathsf{Prot}[\mathsf{Sys}, i, j](1^\lambda, a[i,j], m)$
$return\ m'$

$\textsc{MsgAu}(\mathsf{Prot}, i, j, m)$
$if\ (\mathsf{Prot} = \mathtt{CheckEntry})$
    $if\ (i = 1)\ \mathsf{events}_2[j] \leftarrow m$
    $if\ (\mathsf{out} = 0)\ \mathsf{events}_{\mathsf{fail}}[j] \leftarrow \bot$
$(\mathsf{state}_{\mathsf{Au}}, m', p, \mathsf{out}) \xleftarrow{r} \mathsf{Prot}[\mathsf{Auditor}, i, j](1^\lambda, \mathsf{state}_{\mathsf{Au}}, m)$
$return\ m'$

$\textsc{MsgMo}(i, j, m)$
$(\mathsf{state}_{\mathsf{Mo}}, m', p, \mathsf{out}) \xleftarrow{r} \mathtt{Inspect}[\mathsf{Monitor}, i, j](1^\lambda, \mathsf{state}_{\mathsf{Mo}}, m)$
$return\ m'$
```

Then the transparency overlay satisfies accountability *if for all PT adversaries* $\mathcal{A}$ *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathbf{Adv}^{trace}_{trans,\mathcal{A}}(\lambda) < \nu(\lambda)$.

We then say that a pledged transparency overlay is *secure* if it satisfies consistency, non-frameability, and accountability.

4.3 A generic pledged transparency overlay

We now present a generic version of a pledged transparency overlay. We begin by introducing algorithms for performing various operations in the overlay, and then describe the interactive protocols from Section 4.1 in terms of these algorithms and the algorithms for a dynamic list commitment (DLC) and a signature scheme $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. For ease of exposition we assume that various objects (snapshots, receipts, etc.) contain only the fields necessary to make the protocol work, but could naturally extend our algorithms to cover more general configurations as well.

To start, an event set $\mathsf{eventset}$ contain (at least) a list of events events; a snapshot $\mathsf{snap} = (c, t, \sigma)$ contains a DLC, timing information, and an unforgeable signature; a receipt $\mathsf{rcpt} = (pk, t, \sigma)$ contains a public key, timing information, and an unforgeable signature; and a log $\mathsf{log} = (\mathsf{snap}, \mathsf{events})$ contains a snapshot and a list of events. We denote these subcomponents using bracket notation; e.g., we use $\mathsf{snap}[c]$, or — where subscripts make it appropriately clear — use c_i to denote $\mathsf{snap}_i[c]$.

```
Sys(eventset)                                               LS(log_LS)
                      ---------- eventset ---------->
                                                            if (isOrdered(eventset[events]) = 0) return fail
                                                            if (time(eventset[events][1]) < time(log[events][max])) return fail
                      <---------- $\overrightarrow{\mathsf{rcpt}}$ ----------   $\mathsf{rcpt} \overset{r}{\leftarrow} \mathsf{FormRcpt}(\mathsf{event})\ \forall \mathsf{event} \in \mathsf{eventset}[\mathsf{events}]$
if $\exists(\mathsf{event}, \mathsf{rcpt})$ s.t. $(\mathsf{CheckRcpt}(\mathsf{event}, \mathsf{rcpt}) = 0)$ return 0     $\mathsf{log}_{\mathsf{LS}} \overset{r}{\leftarrow} \mathsf{UpdateLog}(\mathsf{log}_{\mathsf{LS}}, \mathsf{eventset}[\mathsf{events}])$
return 1                                                    return $\varepsilon$
```

Figure 2: The `Log` protocol for pledged transparency overlays.

To perform basic operations on these objects, we also introduce algorithms for forming and verifying snapshots and receipts, and for updating the log. These are defined — with respect to a notion of timing t and a keypair $(pk_{\mathsf{LS}}, sk_{\mathsf{LS}})$ — as follows:

```
$\underline{\mathsf{FormSnap}(c, t)}$
return $(c, t, \mathsf{Sign}(sk_{\mathsf{LS}}, (c, t)))$

$\underline{\mathsf{CheckSnap}(\mathsf{snap})}$
return $\mathsf{Verify}(pk_{\mathsf{LS}}, (\mathsf{snap}[c], \mathsf{snap}[t]), \mathsf{snap}[\sigma])$

$\underline{\mathsf{FormRcpt}(\mathsf{log}, \mathsf{event})}$
return $(pk_{\mathsf{LS}}, t, \mathsf{Sign}(sk_{\mathsf{LS}}, (t, \mathsf{event})))$

$\underline{\mathsf{CheckRcpt}(\mathsf{event}, \mathsf{rcpt})}$
return $\mathsf{Verify}(pk_{\mathsf{LS}}, (\mathsf{rcpt}[t], \mathsf{event}), \mathsf{rcpt}[\sigma])$

$\underline{\mathsf{UpdateLog}(\mathsf{log}, \mathsf{events})}$
$\mathsf{events}' \leftarrow \mathsf{log}[\mathsf{events}] \| \mathsf{events}$
$c' \leftarrow \mathsf{Append}(\mathsf{events}, \mathsf{log}[\mathsf{snap}][c])$
$\mathsf{snap}' \leftarrow \mathsf{FormSnap}(c', t)$
return $(\mathsf{snap}', \mathsf{events}')$
```

Briefly, in the `Log` protocol (Figure 2), an event set is given as input to the actor(s) in the system; this is created by `GenEventSet`, which we describe for our individual applications in Sections 5 and 6 but leave here as an abstract interaction. This event set is sent to the log server, who first checks if it is well formed. The log server then provides a receipt for every event in the set, and sends the receipts back to the system actor(s). If any of the receipts are invalid, the system rejects the interaction, and otherwise it accepts. Either way, the log server updates the log; in our protocol specification here, the log server updates the log immediately, but we discuss in Section 5.3 how this process can be batched and the promises of the log server altered accordingly.

Next, in the `CheckEntry` protocol (Figure 3), some actor in the system sends an event and a receipt to the auditor, who first checks that the receipt is valid. If it is, then the auditor checks if the event already falls within its current purview; i.e., if it falls within the log that the auditor already knows about (according to its snapshot). If it does, then the auditor skips to asking the log server for a proof of inclusion of this event; if not, the auditor must update its snapshot and check that the new snapshot is consistent with the old one. Once the auditor has the proof of inclusion (either after updating or not), it returns to the client whether or not the proof verifies; the client returns whatever it receives from the auditor, and the auditor returns $b = 0$ if the protocol has failed in some way (i.e., the updated snapshot was inconsistent with the old one) and $b = 1$ otherwise.

Next, in the `Inspect` protocol (Figure 4), the monitor sends its current snapshot to the log server, and the log server responds with all events that have been logged since then, along with an updated snapshot. If this list of appended events is valid (i.e., ordered and consistent with the new snapshot), the monitor can update its records and look for any bad events in this new list. It returns $b = 1$ if the protocol has gone smoothly; i.e., if the new list and snapshot seem to have been formed appropriately.

Finally, in the `Gossip` protocol (Figure 5), the auditor and monitor begin by exchanging snapshots, and by ensuring that each snapshot is valid. The monitor then attempts to demonstrate any inconsistencies between

```
     Sys(event)          Auditor(snap_Au, events_bad)                    LS(log_LS)

1        --- event, rcpt --->
2        <--- b ---   $b \leftarrow \mathsf{CheckRcpt}(\mathsf{event}, \mathsf{rcpt})$
3                     if $(b = 0)$ return fail
4                     update $\leftarrow (\mathsf{rcpt}[t] > t_{\mathsf{Au}})$
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(if update)
5                                     --- $\mathsf{snap}_{\mathsf{Au}}$ --->
6                                     <--- $\mathsf{snap}_{\mathsf{LS}}, \pi$ ---   $\pi \leftarrow \mathsf{ProveAppend}(c_{\mathsf{Au}}, c_{\mathsf{LS}}, \mathsf{events}_{\mathsf{LS}})$
7        <--- b ---   $b \leftarrow \mathsf{CheckSnap}(\mathsf{snap}_{\mathsf{LS}}) \wedge \mathsf{CheckAppend}(c_{\mathsf{Au}}, c_{\mathsf{LS}}, \pi)$
8                     if $(b = 0)$ return 0
9                     $\mathsf{snap}_{\mathsf{Au}} \leftarrow \mathsf{snap}_{\mathsf{LS}}$
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
10       <--- b ---   $b \leftarrow (\mathsf{rcpt}[t] \leq t_{\mathsf{Au}})$
11                    if $(b = 0)$ return fail
12                                    --- $\mathsf{event}, \mathsf{snap}_{\mathsf{Au}}$ --->
13                                    <--- $\pi'$ ---   $\pi' \leftarrow \mathsf{ProveIncl}(c_{\mathsf{Au}}, \mathsf{event}, \mathsf{events}_{\mathsf{LS}})$
14       <--- b ---   $b \leftarrow \mathsf{CheckIncl}(c_{\mathsf{Au}}, \mathsf{event}, \pi')$          return $\varepsilon$
15   return b         if $(b = 0)$ $\mathsf{events}_{\mathsf{bad}} \leftarrow \mathsf{events}_{\mathsf{bad}} \| (\mathsf{event}, \mathsf{rcpt})$
16                    return 1
```

Figure 3: The `CheckEntry` protocol for pledged transparency overlays. The parts of the protocol that may not be carried out (depending on the 'if' clause) are marked with dashed lines.

the two snapshots (i.e., demonstrate that they represent forking or distinct logs) and — if any inconsistencies do exist — this is returned as evidence of the log server's misbehavior.

To augment the protocol for pledged overlays, we include in Figure 5 a further optional interaction in which the auditor sends to the monitor all events for which the `CheckEntry` protocol failed, to see if they are being monitored; these are stored in a list $\mathsf{events}_{\mathsf{bad}}$ that is now part of the auditor's state and updated in the `CheckEntry` protocol (line 15 of Figure 3). This allows the auditor and monitor to detect and provide evidence for the additional type of misbehavior in which the log server simply drops events from the log. This means that the auditor and monitor can provide two types of evidence: evidence that the log server presented them with forked or distinct views of the log, or evidence that the log server reneged on the promise it gave in a receipt. We thus instantiate the algorithm CheckEvidence as follows:

```
$\underline{\mathsf{CheckEvidence}(pk_{\mathsf{LS}}, \mathsf{evidence})}$
if $(\mathsf{evidence} = \perp)$ return 0
$(\mathsf{snap}_1, \mathsf{snap}_2, (\mathsf{event}, \mathsf{rcpt}), \pi) \leftarrow \mathsf{evidence}$
if $(\mathsf{CheckSnap}(\mathsf{snap}_1) = 0)$ return 0
if $(\mathsf{CheckSnap}(\mathsf{snap}_2) = 0)$ return 0
if $((\mathsf{event}, \mathsf{rcpt}) = (\perp, \perp))$ return $(\mathsf{CheckInconsistent}(c_1, t_1, c_2, \pi) \wedge (t_1 \leq t_2))$
return $(\mathsf{CheckRcpt}(\mathsf{event}, \mathsf{rcpt}) \wedge \mathsf{CheckNotIncl}(c_2, \mathsf{event}, \pi) \wedge (\mathsf{rcpt}[t] \leq t_2))$
```

Finally, our gossip protocol assumes the monitor has a more up-to-date snapshot than the auditor, which protects against an adversarial log server trivially winning the consistency game (Definition 4.1) by ignoring the monitor. One could also imagine a protocol in which the monitor pauses, updates (using the `Inspect` protocol), and then resumes its interaction with the auditor, in which case the extra winning condition in Definition 4.1 could be dropped.

Theorem 4.4. *If the DLC is secure in the augmented setting and the signature scheme is unforgeable (i.e., EUF-CMA secure), then the protocols presented in Figures 2-5 and the algorithms presented above comprise a secure pledged transparency overlay, as defined in Section 4.2.*

A proof of this theorem can be found in Appendix C. Briefly, consistency follows from three properties of the dynamic list commitment: provable inconsistency, append-only, and soundness. Together, these ensure

```
$\underline{\mathsf{Monitor}(\mathsf{snap}_{\mathsf{Mo}}, \mathsf{events}_{\mathsf{bad}}, \mathsf{events}_{\mathsf{Mo}})}$                    $\underline{\mathsf{LS}(\mathsf{log})}$

                    $\xrightarrow{\mathsf{snap}_{\mathsf{Mo}}}$
                                                  $j \leftarrow \min\{i \mid \mathsf{CheckCom}(c_{\mathsf{Mo}}, \mathsf{events}[1:i])\}$
                    $\xleftarrow{\mathsf{snap}, \mathsf{events}_{\triangle}}$          $\mathsf{events}_{\triangle} \leftarrow \mathsf{events}[j+1:\,]$
if $(\mathsf{CheckSnap}(\mathsf{snap}) = 0) \vee (\mathsf{isOrdered}(\mathsf{events}_{\triangle}) = 0)$ return 0          return $\varepsilon$
    if $(\mathsf{time}(\mathsf{events}_{\triangle}[1]) < t_{\mathsf{Mo}})$ return 0
    $c' \leftarrow \mathsf{Append}(\mathsf{events}_{\triangle}, c_{\mathsf{Mo}})$
    if $(c' \neq \mathsf{snap}[c])$ return 0
    $\mathsf{snap}_{\mathsf{Mo}} \leftarrow \mathsf{snap}; \mathsf{events}_{\mathsf{Mo}} \leftarrow \mathsf{events}_{\mathsf{Mo}} \| \mathsf{events}_{\triangle}$
    update $\mathsf{events}_{\mathsf{bad}}$ using out-of-band checks
    return 1
```

Figure 4: The `Inspect` protocol.

that if the log server presents an inconsistent view of the log to the auditor and monitor, then the commitment seen by the auditor in its snapshot — which, crucially, was updated using ProveAppend and used to demonstrate the inclusion of events — is inconsistent with the list seen by the monitor. By provable inconsistency, the monitor can thus provide a proof of inconsistency that comprises valid evidence of the log server's misbehavior. Non-frameability, on the other hand, follows from the unforgeability of the signature scheme and from the difficulty of forging either a proof of inconsistency or a proof of non-inclusion. Finally, accountability follows from the provable non-inclusion of the DLC, as if an event is missing from the log then the auditor and monitor should be able to provide valid evidence of this (in the form of a receipt promising to include a given event and a proof of non-inclusion of that event).

4.4 A generic basic transparency overlay

A basic transparency overlay is essentially a simpler version of a pledged transparency overlay, so we do not give a full description of the protocols here, but instead describe the necessary modifications that must be made.

The most obvious modification is that all of the parts that involve receipts do not exist in the basic variant. Thus, the `Log` protocol for a basic transparency overlay omits lines 4-5 from Figure 2 but otherwise remains the same. Next, in the `CheckEntry` protocol, the auditor now cannot use the receipt to check if it needs to update, so it must use time(event) instead. The `Inspect` protocol contains no mention or usage of receipts, and thus is exactly the same in the basic variant. This leaves the `Gossip` protocol, in which the only significant modification is that basic transparency overlays cannot provide the second type of evidence (in which the auditor and monitor use the receipt to prove that the log server promised to include an event but then did not), so do not attempt to produce it (lines 9-12 of Figure 5). This also means that evidence consists only of the two snapshots and a proof. Using this simpler form of evidence, the CheckEvidence algorithm for basic overlays then runs as follows:

```
$\underline{\mathsf{CheckEvidence}(pk_{\mathsf{LS}}, \mathsf{evidence})}$
if $(\mathsf{evidence} = \bot)$ return 0
$(\mathsf{snap}_1, \mathsf{snap}_2, \pi) \leftarrow \mathsf{evidence}$
if $(\mathsf{CheckSnap}(\mathsf{snap}_1) = 0)$ return 0
if $(\mathsf{CheckSnap}(\mathsf{snap}_2) = 0)$ return 0
return $(\mathsf{CheckInconsistent}(c_1, t_1, c_2, \pi) \wedge (t_1 \leq t_2))$
```

As the basic transparency overlay thus involves only minor modifications to the pledged transparency overlay, we do not prove its security from scratch, but instead prove the following theorem as a special case of Theorem 4.4.

Theorem 4.5. *If the DLC is secure in the augmented setting and the signature scheme is unforgeable (i.e., EUF-CMA secure), then the modified protocols and algorithms described above comprise a secure basic transparency overlay, as defined in Section 4.1.*

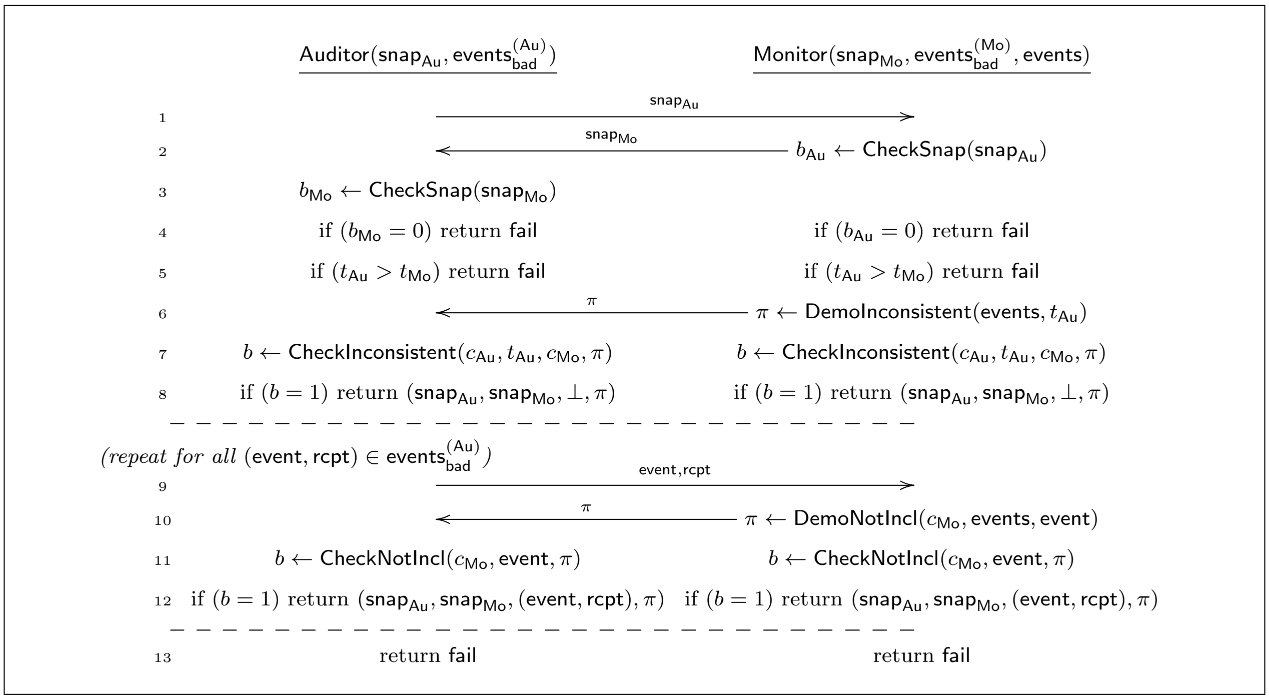

$\underline{\mathsf{Auditor}(\mathsf{snap}_{\mathsf{Au}}, \mathsf{events}^{(\mathsf{Au})}_{\mathsf{bad}})}$
$\underline{\mathsf{Monitor}(\mathsf{snap}_{\mathsf{Mo}}, \mathsf{events}^{(\mathsf{Mo})}_{\mathsf{bad}}, \mathsf{events})}$
$\xrightarrow{\mathsf{snap}_{\mathsf{Au}}}$
$\xleftarrow{\mathsf{snap}_{\mathsf{Mo}}}$ $b_{\mathsf{Au}} \leftarrow \mathsf{CheckSnap}(\mathsf{snap}_{\mathsf{Au}})$
$b_{\mathsf{Mo}} \leftarrow \mathsf{CheckSnap}(\mathsf{snap}_{\mathsf{Mo}})$
if $(b_{\mathsf{Mo}} = 0)$ return fail | if $(b_{\mathsf{Au}} = 0)$ return fail
if $(t_{\mathsf{Au}} > t_{\mathsf{Mo}})$ return fail | if $(t_{\mathsf{Au}} > t_{\mathsf{Mo}})$ return fail
$\xleftarrow{\pi}$ $\pi \leftarrow \mathsf{DemoInconsistent}(\mathsf{events}, t_{\mathsf{Au}})$
$b \leftarrow \mathsf{CheckInconsistent}(c_{\mathsf{Au}}, t_{\mathsf{Au}}, c_{\mathsf{Mo}}, \pi)$ | $b \leftarrow \mathsf{CheckInconsistent}(c_{\mathsf{Au}}, t_{\mathsf{Au}}, c_{\mathsf{Mo}}, \pi)$
if $(b = 1)$ return $(\mathsf{snap}_{\mathsf{Au}}, \mathsf{snap}_{\mathsf{Mo}}, \bot, \pi)$ | if $(b = 1)$ return $(\mathsf{snap}_{\mathsf{Au}}, \mathsf{snap}_{\mathsf{Mo}}, \bot, \pi)$
(repeat for all $(\mathsf{event}, \mathsf{rcpt}) \in \mathsf{events}^{(\mathsf{Au})}_{\mathsf{bad}}$*)*
$\xrightarrow{\mathsf{event}, \mathsf{rcpt}}$
$\xleftarrow{\pi}$ $\pi \leftarrow \mathsf{DemoNotIncl}(c_{\mathsf{Mo}}, \mathsf{events}, \mathsf{event})$
$b \leftarrow \mathsf{CheckNotIncl}(c_{\mathsf{Mo}}, \mathsf{event}, \pi)$ | $b \leftarrow \mathsf{CheckNotIncl}(c_{\mathsf{Mo}}, \mathsf{event}, \pi)$
if $(b = 1)$ return $(\mathsf{snap}_{\mathsf{Au}}, \mathsf{snap}_{\mathsf{Mo}}, (\mathsf{event}, \mathsf{rcpt}), \pi)$ | if $(b = 1)$ return $(\mathsf{snap}_{\mathsf{Au}}, \mathsf{snap}_{\mathsf{Mo}}, (\mathsf{event}, \mathsf{rcpt}), \pi)$
return fail | return fail

Figure 5: The `Gossip` protocol for pledged transparency overlays. The optional part of the protocol is marked with dashed lines.

Proof. (Sketch.) Correctness and compact auditability follow from the same arguments as in the proof of Theorem 4.4. The proof of consistency is nearly identical, with the main differences being that there are fewer cases to rule out if $\mathsf{CheckEvidence}(pk_{\mathsf{LS}}, \mathsf{evidence}) = 0$ (as only one type of evidence can be produced), and that the maps Q_{Au} and Q_{Mo} used in the proof store slightly different values.

Similarly, the proof of non-frameability is essentially a shortened version of the proof of Theorem 4.4. In particular, the lack of receipts in the basic setting means that some of the bad events introduced in this proof are irrelevant, so we need only consider events E_1 (the event in which a proof of inconsistency is faked) and E_2 (the event in which a snapshot is forged). The reductions that bound the probability of these events are nearly identical to the reductions in the proof of Equation 3 and 4 respectively. □

5 Certificate Transparency

In this section, we describe how CT instantiates a pledged transparency overlay (as defined formally in Section 4.2), discuss how the formal notions of overlay security imply more intuitive notions of security specific to the setting of issuing certificates, and finally discuss the requirements of a practical deployment of CT.

5.1 CT is a secure pledged overlay

As depicted in Section 2, Certificate Transparency has three actors in the system Sys: a certificate authority CA, a website owner Site, and a client Client. One of the first two actors must participate in the `Log` protocol,[3] to ensure that the certificate issued by CA to Site ends up in the log, and the client participates in the `CheckEntry` protocol to check that the certificate presented to it by a website is in the log.

In the parlance of CT, an event is a (basic) certificate $\mathsf{cert} = (pk_{\mathsf{name}}, \sigma_{\mathsf{CA}})$, where σ_{CA} is the CA's signature on the site's public key pk_{name},[4] a receipt is a *signed certificate timestamp* (SCT), and a snapshot is a *signed tree head* (STH). For the notion of timing needed for snapshots and receipts, one could pick the current local time of the log server and either use this value directly as t or incorporate into it some buffer period, which

[3]This means that either the website obtains the signed certificate from the CA and then goes on to enter it into the log, or the CA signs the certificate and enters it into the log before returning the extended certificate to the website.

[4]For simplicity, we include here only the most basic version of the information that needs to be checked for and included in a certificate.

is referred to in the CT documentation as the *maximum merge delay* (MMD). We discuss this further in Section 5.3. Finally, CT instantiates `GenEventSet` as follows:

$$
\begin{array}{ccc}
\underline{\mathsf{Site}(pk_{\mathsf{name}})} & & \underline{\mathsf{CA}} \\
 & \xrightarrow{pk_{\mathsf{name}}} & \\
 & & \sigma \overset{r}{\leftarrow} \mathsf{Sign}(sk_{\mathsf{CA}}, pk_{\mathsf{name}}) \\
 & \xleftarrow{\mathsf{cert}} & \mathsf{cert} \leftarrow (pk_{\mathsf{name}}, \sigma) \\
\text{return } \{\mathsf{cert}\} & & \text{return } \{\mathsf{cert}\}
\end{array}
$$

The rest of the protocols needed for the transparency overlay can be instantiated exactly as in Section 4.3, so Theorem 4.4 carries over directly and we can see that CT provides a secure pledged transparency overlay.

5.2 Further security implications

We have just demonstrated that CT provides a secure transparency overlay, but it is not clear what this means for the specific setting of certificate issuance. To explore this, we first remind ourselves of the security of the underlying system (i.e., the issuance of basic certificates), in which (1) it should be difficult to produce a basic certificate without contacting the CA, and (2) an honest client should accept only (basic) certificates that verify. These are clearly satisfied assuming the correctness and unforgeability of the signature scheme.

Combining the underlying issuance security with the security of the overlay, we can argue that three more intuitive security goals are largely satisfied. First, **an *extended* certificate (i.e., a certificate augmented with an SCT) should not pass verification if it has not been jointly produced by the CA and log server.** This holds because the underlying issuance security implies that it is difficult to produce cert without the CA, and non-frameability implies that it is difficult to produce rcpt without the log server, so it should be difficult to produce $(\mathsf{cert}, \mathsf{rcpt})$ without both the CA and the log server.

Second, **honest clients shouldn't accept "bad" certificates; i.e., certificates that are either improperly formatted or not being monitored.** The underlying issuance security says that if cert does not verify then the client won't accept. Following this, the honest client accepts only if the auditor indicates that the certificate is in the log. By consistency, the auditor's view of the log is consistent with the monitor's view from the last time they engaged in the `Gossip` protocol (unless evidence has been produced to the contrary, at which point we can assume the auditor ceases communication with the log server). If the certificate is older than this, then the certificate is definitely being monitored; if it is newer, then it is not guaranteed that the certificate is being monitored, but if it is not then the auditor can at least detect this during its next iteration of the `Gossip` protocol. Thus, honest clients never accept improperly formatted certificates, and are unlikely to accept unmonitored certificates provided that the auditor and monitor are engaging in the `Gossip` protocol with sufficient frequency.

Finally, **if a log server is misbehaving by omitting certificates from the log that it promised to include, it should be possible to blame it.** If a log server refuses to answer queries, then there is little we can do about this in the context of our overlay (although in a practical setting with more than one log server this problem could be mitigated). If a log server does answer, then it can be formally blamed by accountability, as the SCT acts as non-repudiable evidence that the log server has promised to include a certificate and the corresponding proof of non-inclusion demonstrates that it has not done so.

5.3 Practical considerations

Finally, we discuss some necessary alterations that would be made to our protocol if used in a real deployment.

Batched additions to the log. In Figure 2, the log server currently updates the log during the `Log` protocol, and as a result includes the exact current time in the SCT. To avoid doing this operation every time, this process would be batched, so the time in the SCT would instead be some time in the near future (e.g., the end of the current day). This gap between the current and promised times is referred to in the CT documentation as the *maximum merge delay* (MMD).

Collapsing the overlay into the system. As discussed in the CT documentation, in a real deployment we expect auditors to interact with many different log servers (as the certificates seen by clients may be logged in many different places), but expect monitors to focus on one log and the certificates it contains. There are therefore two possible models: in one, the auditors and monitors are operated as separate services, and monitors can even be used as backup log servers. In the other, the role of the auditor could collapse into the client (e.g., it could be run as a browser extension and responses could be cached), and the role of the monitor could collapse (at least partially) into the website, who could monitor the log to at least keep track of its own certificates.

Privacy concerns. While SSL certificates are public and thus storing them in a public log presents no privacy concern, information might be revealed about individual users through the certificates queried by the auditor (to both the log server and monitor), as well as the choice of signed tree heads and SCTs. We view this as an interesting area for future research, but mention briefly that some of these concerns can be mitigated — with minimal effect on the security of the transparency overlay — by omitting the optional part of Figure 5, in which the auditor reveals to the monitor some of the certificates that it has seen.

6 Amplifying Bitcoin's Security

Although Bitcoin already provides a large degree of transparency — as its transaction ledger, called the blockchain, is globally visible — it does not satisfy the requirements of a transparency overlay. In particular, the miners, who play a role analogous to the log server in producing the blockchain, are not known entities and thus cannot be held responsible; this in turn means that consistency and non-frameability cannot be satisfied. In this section, we thus begin by presenting in Section 6.1 a secure basic transparency overlay for Bitcoin.

One might naturally wonder whether such a distinction is purely pedantic; i.e., if overlaying transparency on top of a transparent system provides any actual benefits. To answer this question in the affirmative, we discuss in Section 6.2 the benefits (in terms of both security and efficiency) that are achieved by applying the transparency overlay. In particular, we show that the addition of a secure transparency overlay relieves regular Bitcoin users (i.e., users wishing only to spend and receive bitcoins) from having to store and verify the entire Bitcoin blockchain, which as of this writing is over 80GB.[5] To go even further, we argue that if one is willing to adopt a distributed rather than a fully decentralized solution (i.e., if one is willing to trust any set of named parties), then the entire Bitcoin system collapses into a CT-like transparency overlay and the need for hash-based mining is eliminated.

6.1 A transparency overlay for Bitcoin

As depicted in Section 2, Bitcoin has three actors in the system Sys: a sender Sender, a receiver $\mathsf{Receiver}$, and a miner Miner. The sender and the miner must participate in the `Log` protocol to enter transactions into the log (although really this can be done by only the miner, after it has collected all relevant transactions), and the receiver participates in the `CheckEntry` protocol to check that the transaction in which it should be receiving bitcoins is in the log. Our transparency overlay for Bitcoin then instantiates `GenEventSet` as follows:

$$\begin{array}{lcc}
\underline{\mathsf{Receiver}} & \underline{\mathsf{Sender}(\mathsf{tx})} & \underline{\mathsf{Miner}(\mathsf{head_{old}}, \mathsf{hgt}, \mathsf{txset})} \\
\text{return } \varepsilon & \multicolumn{2}{c}{\xrightarrow{\quad\quad\quad\mathsf{tx}\quad\quad\quad}} \\
 & \text{return } \varepsilon & \mathsf{txset} \leftarrow \mathsf{txset} \cup \{\mathsf{tx}\} \\
 & & \mathsf{head} \xleftarrow{r} \mathsf{Mine}(\mathsf{head_{old}}, \mathsf{txset}) \\
 & & \text{return } (\mathsf{head}, \mathsf{hgt}, \mathsf{txset})
\end{array}$$

An event is a *transaction* tx, which must have a certain structure (i.e., lists of input and output addresses) and satisfy certain requirements (i.e., that is does not represent double-spending). A set of events $\mathsf{eventset}$ is a *block*, which contains not only a list of transactions txset but also a hash head, a pointer $\mathsf{head_{prev}}$ to the previous block, and a height hgt; combining events in an event set also allows us to impose the required notion of timing, which is the block height hgt. By combining `GenEventSet` with the modified protocols described in Section 4.4, we can thus apply Theorem 4.5 to get a secure basic transparency overlay in the setting of Bitcoin.

6.2 Further security implications

By applying a transparency overlay to Bitcoin, we have provided a method for achieving provable transparency guarantees in this setting. We have also achieved (in a manner similarly observed by Miller et al. [27], although they did not provide any security guarantees) a much more efficient version of the system: senders and receivers

[5]`https://blockchain.info/charts/blocks-size`

	Bitcoin	Naïve overlay	CT-like overlay
Hashing	yes	yes	no
Set of miners	decentralized	hybrid*	distributed
Broadcast	yes	yes	no
Provable security	no	yes*	yes

Table 1: The different tradeoffs between Bitcoin, our naïve overlay, and a "CT-like" overlay in which log servers completely replace miners. Our naïve solution provides the same openness that Bitcoin has for miners but also provable security guarantees for those who make (optional) use of distributed log servers, while our CT-like solution requires trust in the set of log servers but achieves both provable security and significantly better efficiency.

now store nothing (or, if the auditor collapses into the users as discussed in Section 5.3 for CT, they store a snapshot), as compared to the entire blockchain or set of block headers that they were required to store previously. While this goal was of course already achievable by Bitcoin senders and receivers using web solutions (i.e., storing their bitcoins in an online wallet), our system is the first to achieve this goal with any provable security guarantees, thus minimizing the trust that such users must place in any third party.

Our analysis also has implications beyond users' storage of the blockchain. To go beyond our initial attempt at an overlay (which we dub the "naïve overlay" in Table 1), one might observe that the miner provides no additional value beyond that of the log server: whereas in CT the CA was necessary to provide a signature (and more generally is assumed to perform external functions such as verifying the owner of a website), here the miner just collates the transactions and sends them to the log server. By having senders contact log servers directly, one could therefore eliminate entirely the role of mining without any adverse effects on security. Thus, if users are willing to make the trust assumptions necessary for our transparency overlay — namely, to assume that some honest majority of a distributed set of log servers provide the correct response about the inclusion of a transaction — then the system can collapse into a distributed structure (the "CT-like overlay" in Table 1) in which no energy is expended to produce the ledger, and users have minimal storage requirements. Moreover, if users communicate directly with the log server, then we could add a signed acknowledgment from the log server that would allow us to satisfy accountability. Interestingly, this solution closely resembles the recent RSCoin proposal [14] (but with our additional consistency and non-frameability guarantees), which achieves linear scaling in transaction throughput; this provides additional validation and suggests that this distributed approach presents an attractive compromise between the two settings.

7 Conclusions and Open Problems

In this paper, we initiated a formal study of transparency overlays by providing definitions and a generic secure construction of this new primitive. To demonstrate the broad applicability of our generic formalization, we proved that Certificate Transparency (CT) is a secure transparency overlay, and presented a Bitcoin-based transparency overlay that achieves provable notions of security and significantly reduces the storage costs of regular Bitcoin users. Our comparison reveals that in any settings where distributed trust is possible (i.e., one is willing to trust any set of known participants), Bitcoin can collapse into CT and the need for both mining and the storage of the blockchain disappears. On the other hand, if one is not willing to trust anyone, then on a certain level these requirements seem inevitable.

While our constructions provide provably secure properties concerning integrity, it is not clear how our transparency overlay could provide this same value to any system in which a meaningful notion of privacy is required. It is thus an interesting open problem to explore the interaction between transparency and privacy, and in particular to provide a transparency overlay that preserves any privacy guarantees of the underlying system.

Acknowledgments

Sarah Meiklejohn is supported in part by EPSRC Grant EP/M029026/1.

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In G. I. Davida and Y. Frankel, editors, *ISC 2001*, volume 2200 of *LNCS*, pages 379–393,

Malaga, Spain, Oct. 1–3, 2001. Springer, Berlin, Germany.

[2] M. Andrychowicz and S. Dziembowski. Pow-based distributed cryptography with no trusted setup. In *Proceedings of Crypto 2015*, 2015.

[3] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on Bitcoin. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[4] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski. ARPKI: Attack Resilient Public-Key Infrastructure. In *Proceedings of ACM CCS 2014*, pages 382–393, 2014.

[5] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In *Proceedings of PKC 2015*, volume 9020 of *LNCS*, pages 516–538, 2015.

[6] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[7] J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In T. Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285, Lofthus, Norway, May 23–27, 1993. Springer, Berlin, Germany.

[8] I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439, Santa Barbara, CA, USA, Aug. 17–21, 2014. Springer, Berlin, Germany.

[9] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. Research perspectives and challenges for Bitcoin and cryptocurrencies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[10] P. Bright. Independent Iranian hacker claims responsibility for Comodo hack, Mar. 2011.

[11] J. Camenisch, M. Kohlweiss, and C. Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In S. Jarecki and G. Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 481–500, Irvine, CA, USA, Mar. 18–20, 2009. Springer, Berlin, Germany.

[12] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76, Santa Barbara, CA, USA, Aug. 18–22, 2002. Springer, Berlin, Germany.

[13] S. Crosby and D. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[14] G. Danezis and S. Meiklejohn. Centrally banked cryptocurrencies. In *Proceedings of NDSS 2016*, 2016.

[15] B. Dowling, F. Günther, U. Herath, and D. Stebila. Secure logging schemes and Certificate Transparency. In *Proceedings of ESORICS 2016*, 2016.

[16] C. Fromknecht, D. Velicanu, and S. Yakoubov. A decentralized public key infrastructure with identity retention. IACR Cryptology ePrint Archive, Report 2014/803, 2014. `http://eprint.iacr.org/2014/803.pdf`.

[17] J. Garay, A. Kiayias, and N. Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *Proceedings of Eurocrypt 2015*, 2015.

[18] C. Garman, M. Green, and I. Miers. Decentralized anonymous credentials. In *Proceedings of the NDSS Symposium 2014*, 2014.

[19] D. Goodin. Fraudulent Google credential found in the wild, Aug. 2011.

[20] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In *Proceedings of WWW 2013*, pages 679–690, 2013.

[21] B. Laurie, A. Langley, and E. Kasper. Certificate transparency, 2013.

[22] J. Leyden. Inside 'Operation Black Tulip': DigiNotar hack analysed, Sept. 2011.

[23] H. Lipmaa. Secure accumulators from euclidean rings without trusted setup. In F. Bao, P. Samarati, and J. Zhou, editors, *ACNS 12*, volume 7341 of *LNCS*, pages 224–240, Singapore, June 26–29, 2012. Springer, Berlin, Germany.

[24] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: Bringing key transparency to end users. In *Proceedings of USENIX Security 2015*, 2015.

[25] J. Menn. Key Internet operator VeriSign hit by hackers, Feb. 2012.

[26] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 218–238, Santa Barbara, CA, USA, Aug. 20–24, 1989. Springer, Berlin, Germany.

[27] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *Proceedings of POPL 2014*, 2014.

[28] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. `bitcoin.org/bitcoin.pdf`.

[29] Nasdaq. Nasdaq launches enterprise-wide blockchain technology initiative, May 2015.

[30] D. O'Leary, V. D'Agostino, S. R. Re, J. Burney, and A. Hoffman. Method and system for processing Internet payments using the electronic funds transfer network, Nov. 2013.

[31] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. Streaming authenticated data structures. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 353–370, Athens, Greece, May 26–30, 2013. Springer, Berlin, Germany.

[32] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM CCS 08*, pages 437–448, Alexandria, Virginia, USA, Oct. 27–31, 2008. ACM Press.

[33] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *Proceedings of NDSS 2014*, 2014.

A Interactive Protocols

We present a formal description of an interactive protocol between multiple participants, as modified from the definitions given by Bellare and Keelveedhi [5]. Consider a protocol Prot with a set of participants $\mathsf{Parties}$, where each participant is invoked at most q times; then the protocol can be represented as a tuple $(\mathsf{Prot}[\mathsf{party}, j])_{\mathsf{party}\in\mathsf{Parties}, j\in[q]}$. Each algorithm in this tuple is given as input the security parameter 1^λ, a value a, and a message $m \in \{0,1\}^*$, and produces as output a value a', a message m', an indicator party of the next algorithm to run, and an output out, where $\mathsf{out} = \perp$ indicates that the participant has not yet terminated. When all participants have terminated, the protocol has terminated. In the case of a stateful participant, the values a and a' are replaced with $\mathsf{state}_{\mathsf{party}}$.

The execution of a protocol is captured by Run, which takes a list of inputs inputs and returns a list of outputs $\mathsf{outputs}$; the algorithm is described below.

```
$\underline{\mathsf{Run}(1^\lambda, \mathsf{Prot}, \mathsf{Parties}, \mathsf{inputs})}$
$T \leftarrow \emptyset$; $\mathsf{party} \leftarrow \mathsf{Parties}[1]$; $m \leftarrow \varepsilon$; $\mathsf{outputs} \leftarrow \emptyset$
for $\mathsf{party} \in \mathsf{Parties}$ do $a[\mathsf{party}, 1] \leftarrow \mathsf{inputs}[\mathsf{party}]$; $\mathsf{round}[\mathsf{party}] \leftarrow 1$
while $T \neq \mathsf{Parties}$ do
  if $\mathsf{party} \in T$ return $\perp$
  $i \leftarrow \mathsf{round}[\mathsf{party}]$
  $(a[\mathsf{party}, i+1], m, \mathsf{party}', \mathsf{out}) \xleftarrow{r} \mathsf{Prot}[\mathsf{party}, i](1^\lambda, a[\mathsf{party}, i], m)$
  if $(\mathsf{out} \neq \perp)$ $T \leftarrow T \cup \{\mathsf{party}\}$; $\mathsf{outputs}[\mathsf{party}] \leftarrow \mathsf{out}$
  $\mathsf{round}[\mathsf{party}] \leftarrow i+1$; $\mathsf{party} \leftarrow \mathsf{party}'$
return $\mathsf{outputs}$
```

Finally, the transcript (i.e., the messages exchanged during the protocol) is captured by Msgs, which takes a list of inputs and returns a matrix of messages; the algorithm is described below.

```
$\underline{\mathsf{Msgs}(1^\lambda, \mathsf{Prot}, \mathsf{Parties}, \mathsf{inputs})}$
$T \leftarrow \emptyset$; $\mathsf{party} \leftarrow \mathsf{Parties}[1]$; $m \leftarrow \varepsilon$; $M \leftarrow \varepsilon$
for $\mathsf{party} \in \mathsf{Parties}$ do $a[\mathsf{party}, 1] \leftarrow \mathsf{inputs}[\mathsf{party}]$; $\mathsf{round}[\mathsf{party}] \leftarrow 1$
while $T \neq \mathsf{Parties}$ do
  if $\mathsf{party} \in T$ return $\perp$
  $i \leftarrow \mathsf{round}[\mathsf{party}]$
  $(a[\mathsf{party}, i+1], m, \mathsf{party}', \mathsf{out}) \xleftarrow{r} \mathsf{Prot}[\mathsf{party}, i](1^\lambda, a[\mathsf{party}, i], m)$
  if $(\mathsf{out} \neq \perp)$ $T \leftarrow T \cup \{\mathsf{party}\}$
  $\mathsf{round}[\mathsf{party}] \leftarrow i+1$; $\mathsf{party} \leftarrow \mathsf{party}'$; $M[\mathsf{party}][\mathsf{round}[\mathsf{party}]] \leftarrow m'$
return $M$
```

B Hash-Based Dynamic List Commitments

In Section 3.2, we presented the formalization of a dynamic list commitment (DLC), and defined its algorithms and desired security properties abstractly. To prove that such structures exist, we present two constructions here and prove their security.

B.1 An instantiation based on hash chains

Briefly, our first construction is essentially a rolling hash chain: the commitment to a list is an iterated hash of its elements (i.e., a hash of the first element is hashed with the second element, etc.), and as new elements are appended they are folded into the hash. Proofs are simple and mainly involve revealing certain parts of the list and committing to the rest.

```
$\underline{\mathsf{Com}(\mathsf{list})}$
$h \leftarrow \varepsilon$
for all $1 \leq i \leq \mathsf{len}(\mathsf{list})$
  $h \leftarrow H(h \| \mathsf{list}[i])$
$c \leftarrow (h, \mathsf{len}(\mathsf{list}))$
return $c$
```

```
$\underline{\mathsf{Append}(\mathsf{list}_\Delta, c_{\mathsf{old}})}$
$(h, \ell) \leftarrow c_{\mathsf{old}}$
for all $1 \leq i \leq \mathsf{len}(\mathsf{list}_\Delta)$
  $h \leftarrow H(h \| \mathsf{list}_\Delta[i])$
$c \leftarrow (h, \ell + \mathsf{len}(\mathsf{list}_\Delta))$
return $c$
```

```
$\underline{\mathsf{CheckCom}(c, \mathsf{list})}$
return $(c = \mathsf{Com}(\mathsf{list}))$
```

```
$\underline{\mathsf{ProveAppend}(c_{\mathsf{old}}, c_{\mathsf{new}}, \mathsf{list})}$
$(h_{\mathsf{old}}, \ell_{\mathsf{old}}) \leftarrow c_{\mathsf{old}}$
if $c_{\mathsf{new}} \neq \mathsf{Append}(\mathsf{list}[\ell_{\mathsf{old}}+1:], c_{\mathsf{old}})$
  return $\perp$
return $\mathsf{list}[\ell_{\mathsf{old}}+1:]$
```

```
$\underline{\mathsf{CheckAppend}(c_{\mathsf{old}}, c_{\mathsf{new}}, \pi)}$
return $(c_{\mathsf{new}} = \mathsf{Append}(\pi, c_{\mathsf{old}}))$
```

```
$\underline{\mathsf{ProveIncl}(c, \mathsf{elmt}, \mathsf{list})}$
$c' \leftarrow (\varepsilon, 0)$; $j \leftarrow 0$
for all $1 \leq i \leq \mathsf{len}(\mathsf{list})$
  if $(\mathsf{list}[i] = \mathsf{elmt})$ $j \leftarrow i$; break
  $c' \leftarrow \mathsf{Append}(\{\mathsf{list}[i]\}, c')$
if $((j = 0) \vee c \neq \mathsf{Append}(\mathsf{list}[j:], c'))$
  return $\perp$
$\pi' \leftarrow \mathsf{ProveAppend}(\mathsf{Append}(\{\mathsf{elmt}\}, c'), c, \mathsf{list})$
return $(c', \pi')$
```

```
$\underline{\mathsf{CheckIncl}(c, \mathsf{elmt}, \pi)}$
if $(\pi = \perp)$ return $0$
$(c', \pi') \leftarrow \pi$
return $\mathsf{CheckAppend}(\mathsf{Append}(\{\mathsf{elmt}\}, c'), c, \pi')$
```

```
DemoInconsistent(list, $(c', \ell'), t'$)
// form DLC to time before $t'$
$c_{\mathsf{pre}} \leftarrow (\varepsilon, 0)$; $j \leftarrow 0$
for all $1 \le i \le \mathsf{len}(\mathsf{list})$
  if ($\mathsf{time}(\mathsf{list}[i+1]) \ge t'$) break
  $j \leftarrow i$
  $c_{\mathsf{pre}} \leftarrow \mathsf{Append}(\{\mathsf{list}[i]\}, c_{\mathsf{pre}})$
// check for inconsistencies at time $t'$
$c_{\mathsf{test}} \leftarrow c_{\mathsf{pre}}$
for all $j+1 \le i \le \mathsf{len}(\mathsf{list})$
  if ($\mathsf{time}(\mathsf{list}[i]) > t'$) break
  $c_{\mathsf{test}} \leftarrow \mathsf{Append}(\{\mathsf{list}[i]\}, c_{\mathsf{test}})$
  if ($c_{\mathsf{test}} = c'$)
    return $\perp$
return ($c_{\mathsf{pre}}, \mathsf{list}[j+1:]$)
```

```
CheckInconsistent($(c', \ell'), t', (c, \ell), \pi$)
if ($\pi = \perp$) return 0
($c_{\mathsf{test}}, \mathsf{list}$) $\leftarrow \pi$; $j \leftarrow 0$ Melissa: Should we check for list = [] eve
if ($\mathsf{time}(\mathsf{list}[1]) \ge t'$) $\wedge$ ($c_{\mathsf{test}} \ne (\varepsilon, 0)$)
  return 0
for all $1 \le i \le \mathsf{len}(\mathsf{list})$
  if ($\mathsf{time}(\mathsf{list}[i]) > t'$) break
  $j \leftarrow i$
  $c_{\mathsf{test}} \leftarrow \mathsf{Append}(\{\mathsf{list}[i]\}, c_{\mathsf{test}})$
  if ($c_{\mathsf{test}} = c'$)
    return 0
if ($\mathsf{Append}(\mathsf{list}[j+1:], c_{\mathsf{test}}) \ne c$)
  return 0
return 1
```

```
DemoNotIncl(list, elmt)
$j \leftarrow 0$
for all $1 \le i \le \mathsf{len}(\mathsf{list})$
  if ($\mathsf{time}(\mathsf{list}[i+1]) \ge \mathsf{time}(\mathsf{elmt})$) break
  $j \leftarrow i$
for all $j+1 \le i \le \mathsf{len}(\mathsf{list})$
  if ($\mathsf{time}(\mathsf{list}[i]) > \mathsf{time}(\mathsf{elmt})$) break
  if ($\mathsf{list}[i] = \mathsf{elmt}$) return $\perp$
if ($j = 0$) return ($(\varepsilon, 0), \mathsf{list}$)
$c_{\mathsf{pre}} \leftarrow \mathsf{Com}(\mathsf{list}[:j])$
return ($c_{\mathsf{pre}}, \mathsf{list}[j+1:]$)
```

```
CheckNotIncl($c$, elmt, $\pi$)
if ($\pi = \perp$) return 0
($c_{\mathsf{pre}}, \mathsf{list}$) $\leftarrow \pi$
if ($\mathsf{time}(\mathsf{list}[1]) \ge \mathsf{time}(\mathsf{elmt})$) $\wedge$ ($c_{\mathsf{pre}} \ne (\varepsilon, 0)$)
  return 0
if $\mathsf{Append}(\mathsf{list}, c_{\mathsf{pre}}) \ne c$ return 0
for all $1 \le i \le \mathsf{len}(\mathsf{list})$
  if ($\mathsf{time}(\mathsf{list}[i]) > \mathsf{time}(\mathsf{elmt})$) return 1
  if ($\mathsf{list}[i] = \mathsf{elmt}$) return 0
return 1
```

Theorem B.1. *If $H(\cdot)$ is a collision-resistant hash function, then the dynamic list commitment defined above is secure in both the basic (Definitions 3.1 and 3.2) and augmented (Definition 3.4) settings.*

Proof. (Sketch.) To show this, we need to prove our construction satisfies six properties: (1) correctness (both basic and augmented), (2) binding, (3) soundness, (4) append-only, (5) provable inconsistency, (6) provable non-inclusion, (7) unforgeable inconsistency, and (8) unforgeable inclusion. We go through these each in turn.

Correctness. This follows by construction.

Binding. To win the game, an adversary must produce a tuple $(c, \mathsf{list}_1, \mathsf{list}_2)$ such that $\mathsf{CheckCom}(c, \mathsf{list}_1) = 1$, $\mathsf{CheckCom}(c, \mathsf{list}_2) = 1$, but $\mathsf{list}_1 \ne \mathsf{list}_2$. These first two properties imply that the two lists have the same length and the cumulative hash of each of the lists is the same, but the last property implies that at some point the lists contain a different entry. Thus, an adversary that identifies the point at which the list entries diverge but the resulting hashes are the same can output the two differing inputs to the hash (consisting of the respective DLCs thus far and the divergent entries) to break collision resistance.

(Identifying these points of divergence can always be done in time proportional to the length of the lists, so our reductions here and for the rest of the properties do run in polynomial time.)

Soundness. To win the game, an adversary must produce a tuple $(c, \mathsf{list}, \mathsf{elmt}, \pi = (c', \mathsf{list}'))$ such that $\mathsf{CheckCom}(c, \mathsf{list}) = 1$, $\mathsf{CheckIncl}(c, \mathsf{elmt}, \pi) = 1$, and $\mathsf{elmt} \notin \mathsf{list}$. These first two properties imply that the cumulative hash of list is the same as the cumulative hash of c' and $\mathsf{elmt}||\mathsf{list}'$ (i.e., $\mathsf{Append}(\mathsf{elmt}||\mathsf{list}', c')$), but the last property implies that at some point there must be differing inputs, because elmt is never used as an input in the former but is used explicitly as an input in the latter. Thus, an adversary that identifies the point at which the inputs are different but the resulting hash is the same can break collision resistance.

Append-only. To win the game, an adversary must produce a tuple $(c_1, c_2, \mathsf{list}_2, \mathsf{list}')$ such that $\mathsf{CheckCom}(c_2, \mathsf{list}_2) = 1$, $\mathsf{CheckAppend}(c_1, c_2, \mathsf{list}') = 1$, but c_1 isn't a commitment to any prefix of list_2. The first two properties imply that the cumulative hash of list_2 is the same as the cumulative hash of c_1 and list', but the last property implies that at some point in the middle there must be differing inputs, as c_1 is not equal to any of the intermediate values in the cumulative hash of list_2 (by definition of not being a prefix). An adversary that identifies this point can thus output these differing inputs to break collision resistance.

Provable inconsistency. To win the game, an adversary must produce a tuple (c, t, list) such that $\mathsf{isOrdered}(\mathsf{list}) = 1$, the honest proof of inconsistency doesn't verify, c isn't a commitment to any prefix of list, or for any j such that $\mathsf{CheckCom}(c, \mathsf{list}[1 : j]) = 1$, either $\mathsf{time}(\mathsf{list}[j]) > t$ or $j \neq \mathsf{len}(\mathsf{list})$ and $\mathsf{time}(\mathsf{list}[j + 1]) < t$. By construction of $\mathsf{DemoInconsistent}$, the only time it outputs $\perp$ is when the exists i such that $c' = \mathsf{Com}(\mathsf{list}[: i])$ and $\mathsf{time}(\mathsf{list}[i]) \leq t$, which contradicts the winning conditions.

Now, consider $\mathsf{CheckInconsistent}$. If $\mathsf{DemoInconsistent}$ doesn't produce $\perp$, then the first two and the last cases in which it would output 0 clearly won't happen, by definition of $\mathsf{DemoInconsistent}$. Finally, $\mathsf{DemoInconsistent}$ outputs $c_{\mathsf{pre}} = \mathsf{Com}(\mathsf{list}[: j])$, $\mathsf{list}([j + 1 :]$, so by correctness of Append the third case where it outputs 0 will not happen either. Thus, $\mathsf{isConsistent}(c, t, \mathsf{list}) = 1$, which contradicts our initial assumption.

Provable non-inclusion. To win the game, an adversary must produce a tuple $(\mathsf{list}, \mathsf{elmt})$ such that the honest proof of non-inclusion doesn't verify, $\mathsf{elmt} \notin \mathsf{list}$, and list is ordered. The first winning condition implies that if we parse the proof of non-inclusion as $\pi = (c', \mathsf{list}')$ then: (1) $\mathsf{time}(\mathsf{list}'[1]) \geq \mathsf{time}(\mathsf{elmt})$ and $c' \neq \varepsilon$, (2) $\mathsf{Append}(\mathsf{list}', c') \neq \mathsf{Com}(\mathsf{list})$, (3) there exists an i such that $\mathsf{time}(\mathsf{list}'[i]) > \mathsf{time}(\mathsf{elmt})$, or (4) $\mathsf{elmt} \in \mathsf{list}'$. Because the proof is produced honestly, we can rule out options (1) and (2), and combining the honest behavior with the assumption that the list is ordered and applying a similar argument as in the proof of provable inconsistency means we can also rule out option (3). This leaves option (4), which directly contradicts the second winning condition.

Unforgeable inconsistency. To win the game, an adversary must produce a tuple $(c_1, t, \mathsf{list}_2, c_2, \pi)$ such that $\mathsf{CheckCom}(c_2, \mathsf{list}_2)$, $\mathsf{CheckInconsistent}(c_1, t, c_2, \pi) = 1$, and $\mathsf{isConsistent}(c_1, t, \mathsf{list}_2)$. The only way $\mathsf{CheckInconsistent}$ will output 1 is if $c_2 = \mathsf{Append}(\mathsf{list}, c_{\mathsf{old}})$. With all but negliglble probability it must be the case that $\mathsf{list}_2[\mathsf{len}(\mathsf{list}_2) - \mathsf{len}(\mathsf{list}) + 1 :] = \mathsf{list}$ (i.e., that list matches the end of list_2), as if not we can easily break collision resistance of the hash function.

So, let k be the index such that $\mathsf{list} = \mathsf{list}_2[k :]$. Moreover, because $\mathsf{isConsistent}$ outputs 1, we know that either $j = \mathsf{len}(\mathsf{list})$ or there exists some j with $\mathsf{time}(\mathsf{list}_2[j]) \leq t \leq \mathsf{time}(\mathsf{list}_2[j+1])$ such that $c_1 = \mathsf{Com}(\mathsf{list}_2[1 : j])$. Since list_2 is ordered, and $\mathsf{time}(\mathsf{list}_2[k]) = \mathsf{time}(\mathsf{list}[1]) < t$ (because $\mathsf{CheckInconsistent}$ outputs 1), we know $j + 1 > k$, or in other words $j \geq k$). Now consider the commitment obtained in $\mathsf{CheckInconsistent}$ after the $(j-k+1)$-st append operation. This commitment will be such that if we append the last $\mathsf{len}(\mathsf{list}_2)-j$ elements of list_2, we get c_2, but this commitment is not equal to c_1 (otherwise $\mathsf{CheckInconsistent}$ outputs 0). We also know that if we take c_1 and append the last $\mathsf{len}(\mathsf{list}_2) - j$ elements of list_2, we get c_2. Thus, we again can show a hash collision.

Unforgeable non-inclusion. To win the game, an adversary must produce a tuple $(c, \mathsf{list}, \mathsf{elmt}, \pi)$ such that $\mathsf{CheckCom}(c, \mathsf{list}) = 1$, $\mathsf{CheckNotIncl}(c, \mathsf{elmt}, \pi) = 1$, $\mathsf{isOrdered}(\mathsf{list})$ and $\mathsf{elmt} \in \mathsf{list}$. The only way $\mathsf{CheckNotIncl}$ will output 1 is if $\pi = (c_{\mathsf{old}}, \mathsf{list}_\Delta)$ such that $c = \mathsf{Append}(\mathsf{list}_\Delta, c_{\mathsf{old}})$. Note that with all but negliglble probability it must be the case that $\mathsf{list}[\mathsf{len}(\mathsf{list}) - \mathsf{len}(\mathsf{list}_\Delta) + 1 :] = \mathsf{list}_\Delta$ (i.e. that list_Δ matches the end of list); if not we can easily build a reduction which breaks collision resistance of the hash function. Let k be the index such that $\mathsf{list}_\Delta = \mathsf{list}[k :]$. Moreover, we know $\mathsf{elmt} \in \mathsf{list}$; suppose it appears at position j. Now, since list is ordered, and either $\mathsf{time}(\mathsf{list}[k]) = \mathsf{time}(\mathsf{list}[1]) < \mathsf{time}(\mathsf{elmt})$ or $c_{\mathsf{old}} = (\varepsilon, 0)$ (because $\mathsf{CheckInconsistent}$ outputs 1), we know $j > k$). But the $j - k$th element in list_Δ is not elmt, because $\mathsf{CheckInconsistent}$ outputs 1. Thus if this occurs with non-negligible probability, we will be able to produce a hash collision. □

B.2 An instantiation based on Merkle trees

Before we present our construction of DLCs based on Merkle trees, we must introduce some terminology. For a binary tree T, define the *completed subtrees* to be subtrees of T with height t and 2^{t-1} elements, the *maximal completed subtrees* to be those which are not contained within any other completed subtrees, and the *membership proof* for a node in T to be the siblings of all the ancestors of that node. Then the algorithms required for a DLC are instantiated as follows:[6]

$\underline{\mathsf{Com}(\mathsf{list})}$

Form a binary tree of height $\lceil \log(\mathsf{len}(\mathsf{list})) \rceil + 1$ whose leaves, ordered from left to right, are the elements in list, and where the leaves get closer to the root going from left to right (or stay at the same distance).

For each internal node, compute the hash of its children. Let r be the value at the root. Output $c = (r, \mathsf{len}(\mathsf{list}))$, and output the values at the root of each maximal completed subtree, again going from

[6]To achieve efficiency, we alter the algorithms from their original form in Section 3.2 to allow for an extra input/output aux, which is used to keep track of the internal hashes in the tree.

left to right, as the auxiliary information aux. Note that the positions of these nodes are fully defined by $\mathsf{len}(\mathsf{list})$.

$\mathsf{Append}(\mathsf{list}_\Delta, \mathsf{aux}_{\mathsf{old}}, c_{\mathsf{old}})$

Parse $c_{\mathsf{old}} = (r_{\mathsf{old}}, L_{\mathsf{old}})$; if this is not consistent with aux, output $\perp$. Otherwise, construct the Merkle tree for the new list using the values specified in $\mathsf{aux}_{\mathsf{old}}$ for the maximal completed subtrees, and let r_{new} be the new root value. Output $c_{\mathsf{new}} = (r_{\mathsf{new}}, L_{\mathsf{old}} + \mathsf{len}(\mathsf{list}_\Delta))$, and the roots of all maximal completed subtrees in the new tree as $\mathsf{aux}_{\mathsf{new}}$.

$\mathsf{CheckCom}(c, \mathsf{list})$

Return $(c = \mathsf{Com}(\mathsf{list}))$.

$\mathsf{ProveAppend}(c_{\mathsf{old}}, \mathsf{aux}_{\mathsf{old}}, c_{\mathsf{new}}, \mathsf{list})$

If c_{old} is not consistent with $\mathsf{aux}_{\mathsf{old}}$, output $\perp$. Otherwise, build a list π_{aux} that contains, for each value in aux, a membership proof for the corresponding node in the tree corresponding to c_{new}. Output $\pi = (\pi_{\mathsf{aux}}, \mathsf{aux}_{\mathsf{old}})$.

$\mathsf{CheckAppend}(c_{\mathsf{old}}, c_{\mathsf{new}}, \pi)$

Parse $\pi = (\pi_{\mathsf{aux}}, \mathsf{aux}_{\mathsf{old}})$. If c_{old} is not consistent with $\mathsf{aux}_{\mathsf{old}}$, output 0. Otherwise, output 1 if each of the membership proofs in π_{aux} verify, and 0 otherwise.

$\mathsf{ProveIncl}(c, \mathsf{elmt}, \mathsf{list})$

Let i be the position of elmt in list, and let π_i be the membership proof for elmt at position i in the tree corresponding to list. Output (i, π_i).

$\mathsf{CheckIncl}(c, \mathsf{elmt}, \pi)$

Parse $c = (r, L)$ and $\pi = (i, \pi_i)$. If $i > L$, output 0. Otherwise, verify the membership proof π for elmt at position i in a tree with L leaves, and output 1 if this verification succeeds (and 0 otherwise).

$\mathsf{DemoInconsistent}(\mathsf{list}, c', t')$

If $\mathsf{isOrdered}(\mathsf{list}) = 0$, output $\perp$. Otherwise, parse $\mathsf{list} = \mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n) \| \mathsf{list}_{\mathsf{post}}$, where $\mathsf{time}(e_1) < t'$, $\mathsf{time}(e_n) > t'$, and $\mathsf{time}(e_i) = t'$ for all i, $2 \le i \le n-1$. (If $\mathsf{time}(\mathsf{list}[1]) \ge t'$, set $\mathsf{list}_{\mathsf{pre}} = []$, or if $\mathsf{time}(\mathsf{list}[\mathsf{len}(\mathsf{list})]) \le t'$, set $\mathsf{list}_{\mathsf{post}} = []$.)

If $\mathsf{list}_{\mathsf{pre}} \neq []$, form $(c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}) \leftarrow \mathsf{Com}(\mathsf{list}_{\mathsf{pre}})$, otherwise set $(c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}) = (\epsilon, \epsilon)$. Form $(c_{\mathsf{post}}, \mathsf{aux}_{\mathsf{post}}) \leftarrow \mathsf{Com}(\mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n))$, and $(c, \mathsf{aux}) \leftarrow \mathsf{Com}(\mathsf{list})$. Compute $\pi_{\mathsf{post}} \leftarrow \mathsf{ProveAppend}(c_{\mathsf{post}}, \mathsf{aux}_{\mathsf{post}}, c, \mathsf{list}_{\mathsf{post}})$.

Output $\pi = (c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}, e_1, \ldots, e_n, c_{\mathsf{post}}, \pi_{\mathsf{post}})$.

$\mathsf{CheckInconsistent}(c', t', c, \pi)$

Parse $\pi = (c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}, e_1, \ldots, e_n, c_{\mathsf{post}}, \pi_{\mathsf{post}})$. Check that $\mathsf{time}(e_1) < t'$ or $c_{\mathsf{pre}} = \epsilon$ and that $\mathsf{time}(e_n) > t'$ or $c_{\mathsf{post}} = c$, verify π_{post}, and output 0 if any of these checks fails.

Let $c_{\mathsf{test}} = c_{\mathsf{pre}}$ and $\mathsf{aux} = \mathsf{aux}_{\mathsf{pre}}$. For all i, $1 \le i \le n$, compute $(c_{\mathsf{test}}, \mathsf{aux}) \leftarrow \mathsf{Append}(e_i, \mathsf{aux}, c_{\mathsf{test}})$ and output 0 if $c_{\mathsf{test}} = c'$. If the loop completes then output $(c_{\mathsf{test}} = c_{\mathsf{post}})$.

$\mathsf{DemoNotIncl}(\mathsf{list}, \mathsf{elmt})$

If $\mathsf{isOrdered}(\mathsf{list}) = 0$, output $\perp$. Otherwise, parse $\mathsf{list} = \mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n) \| \mathsf{list}_{\mathsf{post}}$, where $\mathsf{time}(e_1) < \mathsf{time}(\mathsf{elmt})$, $\mathsf{time}(e_n) > \mathsf{time}(\mathsf{elmt})$, and $\mathsf{time}(e_i) = \mathsf{time}(\mathsf{elmt})$ for all i, $2 \le i \le n-1$. (If $\mathsf{time}(\mathsf{list}[1]) \ge \mathsf{time}(\mathsf{elmt})$, set $\mathsf{list}_{\mathsf{pre}} = []$, or if $\mathsf{time}(\mathsf{list}[\mathsf{len}(\mathsf{list})]) \le \mathsf{time}(\mathsf{elmt})$, set $\mathsf{list}_{\mathsf{post}} = []$.)

If $\mathsf{list}_{\mathsf{pre}} \neq []$, form $(c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}) \leftarrow \mathsf{Com}(\mathsf{list}_{\mathsf{pre}})$, otherwise set $(c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}) = (\epsilon, \epsilon)$. Form $(c_{\mathsf{post}}, \mathsf{aux}_{\mathsf{post}}) \leftarrow \mathsf{Com}(\mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n))$, and $(c, \mathsf{aux}) \leftarrow \mathsf{Com}(\mathsf{list})$. Compute $\pi_{\mathsf{post}} \leftarrow \mathsf{ProveAppend}(c_{\mathsf{post}}, \mathsf{aux}_{\mathsf{post}}, c, \mathsf{list}_{\mathsf{post}})$. Output $\pi = (c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}, e_1, \ldots, e_n, c_{\mathsf{post}}, \pi_{\mathsf{post}})$.

$\mathsf{CheckNotIncl}(c, \mathsf{elmt}, \pi)$

Parse $\pi = (c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}, e_1, \ldots, e_n, c_{\mathsf{post}}, \pi_{\mathsf{post}})$. Check that $\mathsf{time}(e_1) < \mathsf{time}(\mathsf{elmt})$ or $c_{\mathsf{pre}} = \epsilon$, and $\mathsf{time}(e_n) > \mathsf{time}(\mathsf{elmt})$ or $c_{\mathsf{post}} = c$, and that $\mathsf{elmt} \notin (e_2, \ldots, e_{n-1})$. If any of these checks fail, output 0. Otherwise, output $(\mathsf{Append}((e_1, \ldots, e_n), c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}) = c_{\mathsf{post}})$.

Theorem B.2. *If $H(\cdot)$ is a collision-resistant hash function, then the dynamic list commitment defined above is secure in both the basic (Definitions 3.1 and 3.2) and augmented (Definition 3.4) settings.*

Proof. (Sketch.) To show this, we need to prove our construction satisfies six properties: (1) correctness, (2) binding, (3) soundness, (4) append-only, (5) provable inconsistency, (6) provable non-inclusion, (7) unforgeable inconsistency, and (8) unforgeable non-inclusion. We go through these each in turn.

Correctness. This follows by construction.

Binding. This follows from a standard Merkle tree argument. To win the game, an adversary $\mathcal{A}$ must output two different lists that correspond to the same commitment $c = (r, L)$. Since L is the same, both lists must have the same length, and since r is the same the corresponding Merkle trees hash to the same root. Thus, an adversary $\mathcal{B}$ can compute the Merkle tree for each list, and consider the first level (starting from the root) at which at least one node differs between the two trees. That node and its neighbor in both trees can be used to break collision resistance.

Soundness. Again, this follow from a standard argument. To win the game, an adversary must output $(c, \mathsf{list}, \mathsf{elmt}, \pi)$. Parse $\pi = (i, \pi_i)$ and recall that π_i contains the values in the sibling nodes on the path from elmt at position i to the root; thus, given a valid π_i and elmt we can also compute the values for the nodes on the path. Then compute the Merkle tree for list, and consider the first node (starting from the root) at which the node value in the path from elmt differs from the node in the tree for list. The values for that node and its neighbor in the path from elmt and in the tree for list can be used to break collision resistance.

Append-only. To win the game, an adversary must output $(c_1, c_2, \mathsf{list}_2, \pi)$. Parse $\pi = (\pi_{\mathsf{aux}}, \mathsf{aux}_{\mathsf{old}})$ and consider the Merkle tree for list_2; if the adversary wins, then its root must match the value in c_2. If c_1 is not a commitment to a prefix of list_2, then there must be at least one node in $\mathsf{aux}_{\mathsf{old}}$ that does not match the corresponding node in the tree for list_2. In that case we can use the membership proof in π_{aux} for that node (which shows membership with respect to the root in c_2) to find a collision, as in the previous two reductions.

Provable inconsistency. To win the game, an adversary must output (c, t, list). Now, consider the cases where $\mathsf{CheckInconsistent}$ outputs 0: (1) the times of events $e_1, \ldots, e_n$ are wrong; (2) the proof π_{post} does not verify; (3) the final value $c_{\mathsf{test}} \neq c$; and (4) for some i, $c_=c'$.

For (1), since list is ordered, $\mathsf{DemoInconsistent}$ always finds a set of events for which the times are correct. The correctness of $\mathsf{ProveAppend}$ implies that (2) cannot happen, and similarly the correctness of Append implies that (3) cannot happen. Finally, by the correctness of Append, case (4) would imply that $\mathsf{CheckCom}(c', \mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_i)) = 1$, which contradicts the assumption that c' is not a commitment to a prefix of list.

Provable non-inclusion. To win the game, an adversary must output $(\mathsf{list}, \mathsf{elmt})$. Now, consider the cases where $\mathsf{CheckNotIncl}$ outputs 0: (1) the times of events $e_1, \ldots, e_n$ are wrong; (2) $\mathsf{elmt} \in (e_2, \ldots, e_{n-1})$; (3) π_{post} does not verify; and (4) $\mathsf{Append}((e_1, \ldots, e_n), c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}) \neq c_{\mathsf{post}}$.

For (1), since list is ordered, $\mathsf{DemoNotIncl}$ always finds a set of events for which the times are correct. Option (2) clearly contradicts the winning condition that $\mathsf{elmt} \notin \mathsf{list}$. The correctness of $\mathsf{ProveAppend}$ implies that (3) cannot happen, and the correctness of Append implies that (4) cannot happen.

Unforgeable inconsistency. To win the game, an adversary must output $(c_1, t, c_2, \mathsf{list}_2, \pi)$ such that c_1 is a commitment to a prefix of list (call this prefix list_1, and the rest of the list list_Δ), c_2 is a commitment to list_2, the tuple $(c_1, t, \mathsf{list}_2)$ is consistent, but $\mathsf{CheckInconsistent}$ accepts the proof π. Parse $\pi = (c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}, e_1, \ldots, e_n, c_{\mathsf{post}}, \pi_{\mathsf{post}})$.

First, suppose that with non-negligible probability either c_{pre} or c_{post} is not a commitment to a prefix of list_2. That allows us to construct an adversary for the append-only property. (The reduction is immediate and we omit it.) If this is not the case, then there exist lists list' and $\mathsf{list}_{\mathsf{post}}$ such that $\mathsf{list}_2 = \mathsf{list}' \| \mathsf{list}_{\mathsf{post}}$ and $\mathsf{CheckCom}(c_{\mathsf{post}}, \mathsf{list}') = 1$. Let $\mathsf{list}_{\mathsf{pre}}$ be the prefix of list_2 corresponding to c_{pre}. Then by correctness of Append and because $\mathsf{CheckInconsistent}$ accepts, we know $c_{\mathsf{post}} = \mathsf{Com}(\mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n))$.

Now, if with non-negligible probability $\mathsf{list}' \neq \mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n)$, we can break the binding property. (Again, this follows from a straightforward reduction.) Thus, we can assume that $\mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n) = \mathsf{list}'$ and is a prefix of list_2.

Finally, because list_2 is ordered, and because t' is greater than equal to the last element in list_1 and $t' \leq \mathsf{list}_\Delta[1]$ (by consistency), we know that $\mathsf{list}_1 = \mathsf{list}_{\mathsf{pre}} \| e_1, \ldots e_i$ for some i, $1 \leq i \leq n-1$. Then by correctness of Append, we know c_1 will be one of the c_{test} values computed in $\mathsf{CheckInconsistent}$, and $\mathsf{CheckInconsistent}$ will output 0.

Unforgeable non-inclusion. To win the game, an adversary must produce $(c, \mathsf{list}, \mathsf{elmt}, \pi)$ such that $\mathsf{isOrdered}(\mathsf{list})$, $\mathsf{CheckCom}(c, \mathsf{list}) = 1$, and $\mathsf{elmt} \in \mathsf{list}$, but $\mathsf{CheckNotIncl}$ accepts the proof π. Parse $\pi = (c_{\mathsf{pre}}, \mathsf{aux}_{\mathsf{pre}}, e_1, \ldots, e_n, c_{\mathsf{post}}, \pi_{\mathsf{post}})$.

First, suppose that with non-negligible probability either c_{pre} or c_{post} is not a commitment to a prefix of list. This allows us to construct an adversary for the append-only property. (The reduction is immediate and we omit it.) Now, assume this is not the case. Then there exist lists $\mathsf{list}', \mathsf{list}_{\mathsf{post}}$ such that $\mathsf{list} = \mathsf{list}' \| \mathsf{list}_{\mathsf{post}}$ and $c_{\mathsf{post}} = \mathsf{Com}(\mathsf{list}')$.

Let $\mathsf{list}_{\mathsf{pre}}$ be the prefix of list corresponding to c_{pre}. Then by correctness of Append and because $\mathsf{CheckNotIncl}$ accepts, we know $c_{\mathsf{post}} = \mathsf{Com}(\mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n))$.

Now, if with non-negligible probability $\mathsf{list}' \neq \mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n)$, then we can break the binding property. (Again, this follows from a straightforward reduction.) Thus, we can assume that $\mathsf{list}_{\mathsf{pre}} \| (e_1, \ldots, e_n) = \mathsf{list}'$ and is a prefix of list.

Finally, because list is ordered, and because $e_1 < \mathsf{time}(\mathsf{elmt}) < e_n$, we know that $\mathsf{elmt} \in (e_2, \ldots, e_{n-1})$. But then $\mathsf{CheckInconsistent}$ will output 0 because it will have $\mathsf{elmt} \in (e_1, \ldots, e_n)$, which is a contradiction. □

C A Proof of Theorem 4.4

Proof. Correctness and compact auditability follow directly from the correctness of the DLC, signature scheme, and generic algorithms, and from the compactness of the DLC. We now establish consistency, non-frameability, and accountability.

Consistency. To show consistency, we break the Msg oracle in Definition 4.1 down into four oracles: three for each execution of the auditor in the `CheckEntry` protocol (roughly, lines 1-4, 6-9, and 13-16 of Figure 3 respectively) and one for the monitor in the `Inspect` protocol (lines 3-10 of Figure 4). We slightly alter the algorithms for the auditor to fit the Bellare-Keelveedhi framework: first, $\mathsf{state}_{\mathsf{Au}}$ now contains an extra field **inst** to keep track of instances. This field is a vector of length n (where n is the maximum number of instances of the protocol in which $\mathsf{Auditor}$ will engage) and $\mathbf{inst}[j] = (\mathtt{cmd}, \mathsf{event}, \mathsf{rcpt})$, where $\mathtt{cmd} \in \{\mathtt{check}, \mathtt{update}, \mathtt{done}\}$. We also maintain a map Q_{Au} such that $\mathsf{event} \mapsto (c, \mathsf{snap}, \pi, \pi')$ and a map Q_{Mo} such that $t \mapsto (\mathsf{snap}, \mathsf{events})$. These algorithms are now defined as follows:

```
$\mathtt{CheckEntry}[\mathsf{Auditor}, 1, j](1^\lambda, \mathsf{state}_{\mathsf{Au}}, m)$
$(\mathsf{event}, \mathsf{rcpt}) \leftarrow m$
$b \leftarrow \mathsf{CheckRcpt}(\mathsf{event}, \mathsf{rcpt})$
if $(b = 0)$ $\mathbf{inst}[j] \leftarrow (\mathtt{done}, \mathsf{event}, \mathsf{rcpt})$; return $(\mathsf{state}_{\mathsf{Au}}, 0, \mathsf{Sys}, \mathsf{fail})$
if $(\mathsf{rcpt}[t] > t_{\mathsf{Au}})$ $\mathbf{inst}[j] \leftarrow (\mathtt{update}, \mathsf{event}, \mathsf{rcpt})$; return $(\mathsf{state}_{\mathsf{Au}}, \mathsf{snap}_{\mathsf{Au}}, \mathsf{LS}, \perp)$
$\mathbf{inst}[j] \leftarrow (\mathtt{check}, \mathsf{event}, \mathsf{rcpt})$
return $(\mathsf{state}_{\mathsf{Au}}, (\mathsf{event}, \mathsf{snap}_{\mathsf{Au}}), \mathsf{LS}, \perp)$

$\mathtt{CheckEntry}[\mathsf{Auditor}, 2, j](1^\lambda, \mathsf{state}_{\mathsf{Au}}, m)$
$(\mathtt{cmd}, \mathsf{event}, \mathsf{rcpt}) \leftarrow \mathbf{inst}[j]$
if $(\mathtt{cmd} = \mathtt{check})$ return $\mathtt{CheckEntry}[\mathsf{Auditor}, 3, j](1^\lambda, \mathsf{state}_{\mathsf{Au}}, m)$
$(\mathsf{snap}_{\mathsf{LS}}, \pi) \leftarrow m$
$b \leftarrow \mathsf{CheckSnap}(\mathsf{snap}_{\mathsf{LS}}) \wedge \mathsf{CheckAppend}(c_{\mathsf{Au}}, c_{\mathsf{LS}}, \pi)$
if $(b = 0)$ $\mathbf{inst}[j] \leftarrow (\mathtt{done}, \mathsf{event}, \mathsf{rcpt})$; return $\mathsf{state}_{\mathsf{Au}}, 0, \mathsf{Sys}, 0)$
$\mathsf{snap}_{\mathsf{Au}} \leftarrow \mathsf{snap}_{\mathsf{LS}}$
$\mathbf{inst}[j] \leftarrow (\mathtt{check}, \mathsf{event}, \mathsf{rcpt})$
return $(\mathsf{state}_{\mathsf{Au}}, (\mathsf{event}, \mathsf{snap}_{\mathsf{Au}}), \mathsf{LS}, \perp)$

$\mathtt{CheckEntry}[\mathsf{Auditor}, 3, j](1^\lambda, \mathsf{state}_{\mathsf{Au}}, m)$
$(\mathtt{cmd}, \mathsf{event}, \mathsf{rcpt}) \leftarrow \mathbf{inst}[j]$
$b \leftarrow (\mathsf{rcpt}[t] \leq t_{\mathsf{Au}})$
if $(b = 0)$ $\mathbf{inst}[j] \leftarrow (\mathtt{done}, \mathsf{event}, \mathsf{rcpt})$; return $(\mathsf{state}_{\mathsf{Au}}, b, \mathsf{Sys}, \perp)$
$b \leftarrow \mathsf{CheckIncl}(c_{\mathsf{Au}}, \mathsf{event}, m)$
if $(b = 0)$ $\mathsf{events}_{\mathsf{bad}} \leftarrow \mathsf{events}_{\mathsf{bad}} \| (\mathsf{event}, \mathsf{rcpt})$
$\mathbf{inst}[j] \leftarrow (\mathtt{done}, \mathsf{event}, \mathsf{rcpt})$; return $(\mathsf{state}_{\mathsf{Au}}, b, \mathsf{Sys}, 1)$
```

We now consider the winning conditions for the consistency game $\mathsf{G}^{\mathrm{cons}}_{\mathcal{A}}(\lambda)$. First, if $\mathsf{CheckEvidence}(pk_{\mathsf{LS}}, \mathsf{evidence}) = 0$, then there are six possibilities: (1) $\mathsf{evidence} = \perp$, or $\mathsf{evidence} = (\mathsf{snap}_{\mathsf{Au}}, \mathsf{snap}_{\mathsf{Mo}}, (\mathsf{event}, \mathsf{rcpt}), \pi)$ and either

2. $\mathsf{snap}_{\mathsf{Au}}$ does not verify;
3. $\mathsf{snap}_{\mathsf{Mo}}$ does not verify;
4. $\mathsf{event} = \perp$ and $\mathsf{CheckInconsistent}(c_{\mathsf{Au}}, t_{\mathsf{Au}}, c_{\mathsf{Mo}}, \pi) = 0$;
5. $\mathsf{CheckNotIncl}(c_{\mathsf{Mo}}, \mathsf{event}, \pi) = 0$; or

6. $\mathsf{CheckRcpt}(\mathsf{event}, \mathsf{rcpt}) = 0$.

Looking at the protocol in Figure 5, we can see that in fact the middle four options can never happen, as the honest auditor and monitor check these values themselves and output them only in the case that they do verify. Similarly, if the final option occurs, then the honest auditor will not include the pair $(\mathsf{event}, \mathsf{rcpt})$ in $\mathsf{events}_{\mathsf{bad}}^{(\mathsf{Au})}$, so it will never be used in the gossip protocol and thus never output as evidence.

This leaves us with only one remaining possibility: that $\mathsf{evidence} = \perp$. Folding in the other winning conditions, we can express this as the event E in which

$$((\mathsf{evidence} = \perp) \wedge (t_{\mathsf{Mo}} \geq t_{\mathsf{Au}}) \wedge (\mathsf{events}_{\mathsf{pass}} \setminus \mathsf{events}_{\mathsf{Mo}} \neq \emptyset)).$$

We build adversaries $\mathcal{B}_1$, $\mathcal{B}_{2,i}$ for all i, $1 \leq i \leq n$, and $\mathcal{B}_3$ such that

$$\Pr[E] \leq \mathbf{Adv}^{\text{p-cons}}_{\text{dlc},\mathcal{B}_1}(\lambda) + \sum_{i=1}^{n} \mathbf{Adv}^{\text{append}}_{\text{dlc},\mathcal{B}_{2,i}}(\lambda) + \mathbf{Adv}^{\text{sound}}_{\text{dlc},\mathcal{B}_3}(\lambda) \tag{1}$$

Combining this with our argument above, we get

$$\begin{aligned}\mathbf{Adv}^{\text{cons}}_{\text{trans},\mathcal{A}}(\lambda) &= \Pr[\mathsf{G}^{\text{cons}}_{\mathcal{A}}(\lambda)] \\ &= \Pr[E] \\ &\leq \mathbf{Adv}^{\text{p-cons}}_{\text{dlc},\mathcal{B}_1}(\lambda) + \sum_{i=1}^{n} \mathbf{Adv}^{\text{append}}_{\text{dlc},\mathcal{B}_{2,i}}(\lambda) + \mathbf{Adv}^{\text{sound}}_{\text{dlc},\mathcal{B}_3}(\lambda),\end{aligned}$$

from which the theorem follows.

Equation 1
By the time the auditor and monitor engage in the `Gossip` protocol, the maps Q_{Au} and Q_{Mo} will have been populated with (trimmed) transcripts of the interactions in which these parties have engaged, which means Q_{Au} will have a set of keys $\{\mathsf{event}_i\}_{i=1}^{n}$ and Q_{Mo} will have a set of keys $\{t_j\}_{j=1}^{m}$, and the auditor and monitor will have respective state $(\mathsf{snap}_{\mathsf{Au}}, \mathsf{events}_{\mathsf{bad}}^{(\mathsf{Au})})$ and $(\mathsf{snap}_{\mathsf{Mo}}, \mathsf{events}_{\mathsf{bad}}^{(\mathsf{Mo})}, \mathsf{events}_{\mathsf{Mo}})$. We make two initial observations about the values in these maps:

1. If we order the keys for Q_{Mo} and define $(\mathsf{snap}_j, \mathsf{events}_j) \leftarrow Q_{\mathsf{Mo}}[t_j]$, then $\mathsf{events}_{\mathsf{Mo}} = \mathsf{events}_1 \| \dots \| \mathsf{events}_m$, $\mathsf{isOrdered}(\mathsf{events}_{\mathsf{Mo}}) = 1$, and $\mathsf{CheckCom}(c_{\mathsf{Mo}}, \mathsf{events}_{\mathsf{Mo}}) = 1$. This follows from the behavior of the honest monitor and from the correctness of the DLC.
2. If we order the values $(c, \mathsf{snap}_{\mathsf{LS}}, \pi, \pi')$ in Q_{Au} by t_{LS}, then $t_{\mathsf{Au}} = t_{\mathsf{LS},n}$ and $c_{\mathsf{Au}} = c_{\mathsf{LS}}[n]$. This follows by the behavior of the honest auditor (and in particular because $\mathsf{snap}_{\mathsf{LS}}$ gets added to Q_{Au} only if the appropriate checks pass).

We now break down the components of E_2 as follows: first, if $\mathsf{evidence} = \perp$ and $t_{\mathsf{Mo}} \geq t_{\mathsf{Au}}$, then the execution of the protocol in Figure 5 reached line 13, which means that

$$\mathsf{CheckInconsistent}(c_{\mathsf{Au}}, t_{\mathsf{Au}}, c_{\mathsf{Mo}}, \mathsf{DemoInconsistent}(\mathsf{events}_{\mathsf{Mo}}, t_{\mathsf{Au}})) = 0. \tag{2}$$

Based on this, we argue that there must exist a prefix $\mathsf{events}_{\mathsf{Au}}$ of $\mathsf{events}_{\mathsf{Mo}}$ such that $\mathsf{CheckCom}(c_{\mathsf{Au}}, \mathsf{events}_{\mathsf{Au}}) = 1$. Define this as event E_{prefix} and suppose to the contrary that it does not hold; then we construct an adversary $\mathcal{B}_1$ against the provable inconsistency of the DLC as follows (we omit the description of MsgAu and MsgMo, as $\mathcal{B}_1$ executes these honestly):

```
$\underline{\mathcal{B}_1(1^\lambda)}$
$pk_{\mathsf{LS}} \xleftarrow{r} \mathcal{A}^{\text{MsgAu},\text{MsgMo}}(1^\lambda)$
return $(c_{\mathsf{Au}}, t_{\mathsf{Au}}, \mathsf{events}_{\mathsf{Mo}})$
```

By Equation 2, these outputs satisfy the first winning condition of $\mathsf{G}^{\text{p-cons}}_{\mathcal{B}_1}(\lambda)$, and by our first initial observation they also satisfy the last winning condition. Furthermore, if $\neg E_{\text{prefix}}$, then they also satisfy the middle winning condition, meaning $\mathcal{B}_1$ wins the game. Thus, $\Pr[\neg E_{\text{prefix}}] \leq \mathbf{Adv}^{\text{p-cons}}_{\text{dlc},\mathcal{B}_1}(\lambda)$.

Moving on to the last component of E_2, if $\mathsf{events}_{\mathsf{pass}} \setminus \mathsf{events}_{\mathsf{Mo}} \neq \emptyset$, then there exists an event event such that $\mathsf{event} \in \mathsf{events}_{\mathsf{pass}}$ but $\mathsf{event} \notin \mathsf{events}_{\mathsf{Mo}}$. The former property implies that the following conditions hold:

1. $\mathsf{event} \in Q_{\mathsf{Au}}$, so $(c, \mathsf{snap}_{\mathsf{LS}}, \pi', \pi) \leftarrow Q_{\mathsf{Au}}[\mathsf{event}]$ is well defined; and either

2. $\mathsf{snap}_{\mathsf{LS}} = \bot$ and $\mathsf{CheckIncl}(c, \mathsf{event}, \pi') = 1$, or
3. $\mathsf{snap}_{\mathsf{LS}} \neq \bot$ and
   (a) $\mathsf{CheckAppend}(c, c_{\mathsf{LS}}, \pi) = 1$;
   (b) $\mathsf{CheckSnap}(\mathsf{snap}_{\mathsf{LS}}) = 1$; and
   (c) $\mathsf{CheckIncl}(c_{\mathsf{LS}}, \mathsf{event}, \pi') = 1$.

Fix event_i as this event, define $(c_i, \mathsf{snap}_i, \pi'_i, \pi_i) \leftarrow Q_{\mathsf{Au}}[\mathsf{event}_i]$, and assume now that E_{prefix} holds; i.e., that there exists a prefix $\mathsf{events}_{\mathsf{Au}}$ of $\mathsf{events}_{\mathsf{Mo}}$ such that $\mathsf{CheckCom}(c_{\mathsf{Au}}, \mathsf{events}_{\mathsf{Au}}) = 1$.

We would now like to argue that there also exists a prefix events_i of $\mathsf{events}_{\mathsf{Au}}$ such that $\mathsf{CheckCom}(c_i, \mathsf{events}_i) = 1$; call this event $E_{\text{i-prefix}}$. Intuitively, this holds because we have a "path" of append proofs from c_i to $c_n = c_{\mathsf{Au}}$, so if no such prefix exists then we can use at least one of these proofs to violate the append-only property of the DLC.

More formally, observe that $E_{\text{i-prefix}}$ is implied by $\bigwedge_{k=i}^{n} E_{\text{k-prefix}}$; i.e., c_i is a commitment to a prefix events_i of $\mathsf{events}_{\mathsf{Au}}$ if c_k is a commitment to a prefix events_k for all k, $i \leq k \leq n$. Thus, $\neg E_{\text{i-prefix}}$ implies $\bigvee_{k=i}^{n} \neg E_{\text{k-prefix}}$. We proceed in a series of hybrids: for each hop k, we assume that $E_{\ell\text{-prefix}}$ holds for all ℓ, $k+1 \leq \ell \leq n$, and we construct an adversary $\mathcal{B}_{2,k}$ as follows (again, omitting the description of MsgAu and MsgMo, in which $\mathcal{B}_{2,k}$ behaves honestly):

```
$\mathcal{B}_{2,k}(1^\lambda)$
$pk_{\mathsf{LS}} \xleftarrow{r} \mathcal{A}^{\text{MsgAu},\text{MsgMo}}(1^\lambda)$
$(c_j, \mathsf{snap}_j, \pi'_j, \pi_j) \leftarrow Q_{\mathsf{Au}}[\mathsf{event}_j]$ for $j = k, k+1$
if $\mathsf{snap}_{k+1} = \bot$ return $\bot$
find prefix $\mathsf{events}_{k+1}$ of $\mathsf{events}_{\mathsf{Mo}}$
return $(c_k, c_{k+1}, \mathsf{events}_{k+1}, \pi_{k+1})$
```

First, we observe that if the prefix events_{k+1} exists (which it does, assuming E_{prefix} and $E_{\ell\text{-prefix}}$ for all ℓ, $k+1 \leq \ell \leq n$), it can be computed efficiently given $\mathsf{events}_{\mathsf{Mo}}$, so $\mathcal{B}_{2,k}$ does run in polynomial time. Similarly, if these events hold then $\mathsf{CheckCom}(c_{k+1}, \mathsf{events}_{k+1}) = 1$, so the first winning condition of $\mathsf{G}^{\text{append}}_{\mathcal{B}_{2,k}}(\lambda)$ is met. Furthermore, by our conditions above, we know that $\mathsf{CheckAppend}(c_k, c_{k+1}, \pi_{k+1}) = 1$, so the second winning condition is met as well. Finally, if $\neg E_{\text{k-prefix}}$, then the third winning condition is met, so $\Pr[\neg E_{\text{k-prefix}}] \leq \mathbf{Adv}^{\text{append}}_{\text{dlc},\mathcal{B}_{2,k}}(\lambda)$. Putting all the hybrids together, we get that $\Pr[E_{\text{prefix}} \wedge \neg E_{\text{i-prefix}}] \leq \sum_{k=i}^{n} \mathbf{Adv}^{\text{append}}_{\text{dlc},\mathcal{B}_{2,k}}(\lambda)$.

Finally, suppose now that $E_{\text{i-prefix}}$ does hold; i.e., that there exists a prefix events_i of $\mathsf{events}_{\mathsf{Au}}$ (which is itself a prefix of $\mathsf{events}_{\mathsf{Mo}}$) such that $\mathsf{CheckCom}(c_i, \mathsf{events}_i) = 1$. Then we can construct an adversary $\mathcal{B}_3$ to break the soundness of the DLC as follows:

```
$\mathcal{B}_3(1^\lambda)$
$pk_{\mathsf{LS}} \xleftarrow{r} \mathcal{A}^{\text{MsgAu},\text{MsgMo}}(1^\lambda)$
$(c_i, \mathsf{snap}_i, \pi'_i, \pi_i) \leftarrow Q_{\mathsf{Au}}[\mathsf{event}_i]$
return $(c_i, \mathsf{events}_i, \mathsf{event}_i, \pi'_i)$
```

Because $E_{\text{i-prefix}}$ holds, $\mathsf{CheckCom}(c_i, \mathsf{events}_i) = 1$, so the first winning condition of $\mathsf{G}^{\text{sound}}_{\mathcal{B}_3}(\lambda)$ is met. By our conditions for $\mathsf{event} \in \mathsf{events}_{\mathsf{pass}}$, we also know the second winning condition is met. Finally, because $\mathsf{event}_i \notin \mathsf{events}_{\mathsf{Mo}}$ (which is implied by E), it must also be the case that $\mathsf{event}_i \notin \mathsf{events}_i$, so we get that $\Pr[E_{\text{prefix}} \wedge E_{\text{i-prefix}}] \leq \mathbf{Adv}^{\text{sound}}_{\text{dlc},\mathcal{B}_3}(\lambda)$.

What we have now shown is that

$$E = \neg E_{\text{prefix}} \vee (E_{\text{prefix}} \wedge \neg E_{\text{i-prefix}}) \vee (E_{\text{prefix}} \wedge E_{\text{i-prefix}}),$$

and that

$$\Pr[\neg E_{\text{prefix}}] \leq \mathbf{Adv}^{\text{p-cons}}_{\text{dlc},\mathcal{B}_1}(\lambda)$$
$$\Pr[E_{\text{prefix}} \wedge \neg E_{\text{i-prefix}}] \leq \sum_{k=i}^{n} \mathbf{Adv}^{\text{append}}_{\text{dlc},\mathcal{B}_{2,k}}(\lambda)$$
$$\Pr[E_{\text{prefix}} \wedge E_{\text{i-prefix}}] \leq \mathbf{Adv}^{\text{sound}}_{\text{dlc},\mathcal{B}_3}(\lambda)$$

Putting everything together, we get that

$$\begin{aligned}\Pr[E] &= \Pr[\neg E_{\text{prefix}}] + \Pr[E_{\text{prefix}} \wedge \neg E_{\text{i-prefix}}] + \Pr[E_{\text{prefix}} \wedge E_{\text{i-prefix}}] \\ &\leq \mathbf{Adv}^{\text{p-cons}}_{\text{dlc},\mathcal{B}_1}(\lambda) + \sum_{i=1}^{n} \mathbf{Adv}^{\text{append}}_{\text{dlc},\mathcal{B}_{2,i}}(\lambda) + \mathbf{Adv}^{\text{sound}}_{\text{dlc},\mathcal{B}_3}(\lambda),\end{aligned}$$

which establishes the equation.

Non-frameability. We break the Msg oracle in Definition 4.2 down into four oracles: one for the execution of the log server during the `Log` protocol (lines 1-6 in Figure 2), two for its executions during the `CheckEntry` protocol (lines 5-6 and 12-14 of Figure 3 respectively), and one for its execution during the `Inspect` protocol (lines 1-4 of Figure 4). We also alter the protocols slightly to add sets M_{rcpt} and M_{snap} to keep track of, respectively, the receipts formed in the `Log` protocol and the snapshots formed in $\mathsf{UpdateLog}$ (during the `Log` protocol).

Define

$$E_{\text{gd-snap-1}} = (\mathsf{CheckSnap}(\mathsf{snap}) \wedge (\mathsf{snap}[c], \mathsf{snap}[t]) \in M_{\mathsf{snap}}),$$

$E_{\text{gd-snap-2}}$ analogously for snap_2, and

$$E_{\text{gd-rcpt}} = (\mathsf{CheckRcpt}(\mathsf{event}, \mathsf{rcpt}) \wedge (\mathsf{event}, \mathsf{rcpt}[t]) \in M_{\mathsf{rcpt}})).$$

We then define the following four events:

$$E_1 : E_{\text{gd-snap-1}} \wedge E_{\text{gd-snap-2}} \wedge (\mathsf{event}, \mathsf{rcpt} = \bot) \wedge \mathsf{CheckInconsistent}(c_1, t_1, c_2, \pi)) \wedge (t_1 \leq t_2)$$
$$E_2 : \neg E_{\text{gd-snap-1}} \vee \neg E_{\text{gd-snap-2}}$$
$$E_3 : \neg E_{\text{gd-rcpt}}$$
$$E_4 : E_{\text{gd-snap-1}} \wedge E_{\text{gd-snap-2}} \wedge E_{\text{gd-rcpt}} \wedge \mathsf{CheckNotIncl}(c_2, \mathsf{event}, \pi)) \wedge (\mathsf{rcpt}[t] \leq t_2)$$

Now, let $\mathcal{A}$ be a PT adversary playing $\mathsf{G}^{\text{frame}}_{\mathcal{A}}(\lambda)$. We build adversaries $\mathcal{B}_i$ for all i, $1 \leq i \leq 4$, such that

$$\Pr[E_1] \leq \mathbf{Adv}^{\text{uf-cons}}_{\text{dlc},\mathcal{B}_1}(\lambda) \tag{3}$$
$$\Pr[E_2] \leq \mathbf{Adv}^{\text{euf-cma}}_{\text{sig},\mathcal{B}_2}(\lambda) \tag{4}$$
$$\Pr[E_3] \leq \mathbf{Adv}^{\text{euf-cma}}_{\text{sig},\mathcal{B}_3}(\lambda) \tag{5}$$
$$\Pr[E_4] \leq \mathbf{Adv}^{\text{uf-incl}}_{\text{dlc},\mathcal{B}_4}(\lambda) \tag{6}$$

We then have that

$$\begin{aligned}\mathbf{Adv}^{\text{frame}}_{\text{trans},\mathcal{A}}(\lambda) &= \Pr[\mathsf{G}^{\text{frame}}_{\mathcal{A}}(\lambda)] \\ &\leq \sum_{i=1}^{4} \Pr[E_i] \\ &\leq \mathbf{Adv}^{\text{uf-cons}}_{\text{dlc},\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\text{euf-cma}}_{\text{sig},\mathcal{B}_2}(\lambda) + \mathbf{Adv}^{\text{euf-cma}}_{\text{sig},\mathcal{B}_3}(\lambda) + \mathbf{Adv}^{\text{uf-incl}}_{\text{dlc},\mathcal{B}_4}(\lambda),\end{aligned}$$

from which the theorem follows.

Equation 3: faking a proof of inconsistency

$\mathcal{B}_1$ behaves as follows, using a map Q such that $c \mapsto \mathsf{events}$ (and omitting the descriptions of all algorithms in which $\mathcal{B}_1$ honestly follows the protocol specification):

```
$\underline{\mathcal{B}_1(1^\lambda)}$
$Q \leftarrow \emptyset$; $(pk_{\mathsf{LS}}, sk_{\mathsf{LS}}) \xleftarrow{r} \mathsf{KeyGen}(1^\lambda)$
$(\mathsf{snap}_1, \mathsf{snap}_2, (\mathsf{event}, \mathsf{rcpt}), \pi) \xleftarrow{r} \mathcal{A}^{\text{Msg}}(1^\lambda, pk_{\mathsf{LS}})$
return $(\mathsf{snap}_1[c], \mathsf{snap}_1[t], \mathsf{snap}_2[c], Q[c_2], \pi)$

$\underline{\texttt{CheckEntry}[\mathsf{LS}, 1, j](1^\lambda, \mathsf{state}_{\mathsf{LS}}, m)}$
$\pi \leftarrow \mathsf{ProveAppend}(m[c], c_{\mathsf{LS}}, \mathsf{events}_{\mathsf{LS}})$
$Q[c_{\mathsf{LS}}] = \mathsf{events}_{\mathsf{LS}}$
return $(\mathsf{state}_{\mathsf{LS}}, (\mathsf{snap}_{\mathsf{LS}}, \pi), \mathsf{Auditor}, \varepsilon)$

$\underline{\texttt{Inspect}[\mathsf{LS}, 1, j](1^\lambda, \mathsf{state}_{\mathsf{LS}}, m)}$
$\langle$execute lines 2-3 of Figure 4$\rangle$
$Q[c_{\mathsf{LS}}] = \mathsf{events}_{\mathsf{LS}}$
return $(\mathsf{state}_{\mathsf{LS}}, (\mathsf{snap}, \mathsf{events}_\Delta), \mathsf{Monitor}, \varepsilon)$
```

It is clear that the interaction with $\mathcal{B}_1$ is identical to the interaction that $\mathcal{A}$ expects, as $\mathcal{B}_1$ executes all of the protocols honestly. Intuitively, $\mathcal{B}_1$ keeps track of the log list every time it returns a snapshot to the adversary. Because $(c_2, t_2) \in M_{\mathsf{snap}}$, $c_2 \in Q$, so $Q[c_2]$ is well defined. It also holds that $\mathsf{isConsistent}(c_1, t, Q[c_2])$ (again, by the honest behavior of $\mathcal{B}_1$), so the second winning condition of $\mathsf{G}^{\text{uf-cons}}_{\mathcal{B}_1}(\lambda)$ is met.

Furthermore, the honest behavior of $\mathcal{B}_1$ and the correctness of the DLC imply that $c_2 = \mathsf{Com}(Q[c_2])$, so the first winning condition is met. Finally, because E_1 holds, it is the case that $\mathsf{CheckInconsistent}(c_1, t_1, c_2, \pi) = 1$, so the last winning condition is met as well and $\mathcal{B}_1$ wins whenever E_1 holds.

Equation 4: forging a snapshot
$\mathcal{B}_2$ behaves as follows (omitting the description of all algorithms, in which $\mathcal{B}_2$ honestly follows the protocol specification):

```
$\mathcal{B}_2^{\textsc{Sign}}(1^\lambda, pk)$
$Q \leftarrow \emptyset$
$(\mathsf{snap}_1, \mathsf{snap}_2, (\mathsf{event}, \mathsf{rcpt}), \pi) \xleftarrow{r} \mathcal{A}^{\textsc{Msg}}(1^\lambda, pk)$
if $(c_1, t_1) \notin Q$ return $((c_1, t_1), \sigma_1)$
return $((c_2, t_2), \sigma_2)$

$\mathsf{Log}[\mathsf{LS}, 1, j](1^\lambda, \mathsf{state}_{\mathsf{LS}}, m)$
$\langle$execute lines 2-3 of Figure 2$\rangle$
$\mathsf{rcpt} \leftarrow (pk, t, \textsc{Sign}((t, \mathsf{event})))$ for all $\mathsf{event} \in m[\mathsf{events}]$
$\mathsf{events}' \leftarrow \mathsf{log}[\mathsf{events}] \| \mathsf{events}$
$c' \leftarrow \mathsf{Append}(\mathsf{events}, \mathsf{log}[\mathsf{snap}][c])$
$\mathsf{snap}' \leftarrow (c', t, \textsc{Sign}((c', t)))$
$Q \leftarrow Q \cup \{(c', t)\}$
return $(\mathsf{state}_{\mathsf{LS}}, \overrightarrow{\mathsf{rcpt}}, \mathsf{Sys}, \varepsilon)$
```

It is clear that the interaction with $\mathcal{B}_2$ is identical to the interaction that $\mathcal{A}$ expects, as $\mathcal{B}_2$ executes all the algorithms honestly and forms the signatures using its signing oracle. Equally, if E_2 holds then it must be the case that $(c_1, t_1) \notin M_{\mathsf{snap}}$ or $(c_2, t_2) \notin M_{\mathsf{snap}}$ (because the signature included in a receipt is on (t, event) rather than (c, t), so will not pass the snapshot verification), as if verification does not pass then $\mathsf{CheckEvidence}$ outputs 0. If $(c_1, t_1) \notin M_{\mathsf{snap}}$ then (c_1, t_1) was not queried to the $\textsc{Sign}$ oracle, and analogously if $(c_2, t_2) \notin M_{\mathsf{snap}}$ then (c_2, t_2) was not queried to the $\textsc{Sign}$ oracle. Thus, $\mathcal{B}_2$ succeeds whenever E_2 holds.

Equation 5: forging a receipt
$\mathcal{B}_3$ behaves nearly identically to $\mathcal{B}_2$, with the following two differences: (1) it maintains a set Q for the message/signature pairs related to receipt rather than to snapshots, and (2) at the end of the game, it outputs $((t, \mathsf{event}), \sigma)$ (where these values are pulled from rcpt). Again, it clear that the interaction with $\mathcal{B}_3$ is identical to the one that $\mathcal{A}$ expects. Furthermore, if E_3 holds then it must be the case that $\mathsf{event} \notin M_{\mathsf{rcpt}}$, as if $\mathsf{rcpt}[pk] \neq pk$ then $\mathsf{CheckEvidence}$ outputs 0. If $\mathsf{event} \notin M_{\mathsf{rcpt}}$ then $(\mathsf{rcpt}[t], \mathsf{event})$ was not queried to the $\textsc{Sign}$ oracle, so $\mathcal{B}_3$ succeeds whenever E_3 holds.

Equation 6: faking a proof of non-inclusion
Finally, $\mathcal{B}_4$ behaves as follows, using a map Q such that $c \mapsto \mathsf{events}$ (and omitting the descriptions of $\textsc{Msg}$, $\mathsf{Log}[\mathsf{LS}, 1, j]$, $\mathtt{CheckEntry}[\mathsf{LS}, 1, j]$, and $\mathtt{Inspect}[\mathsf{LS}, 1, j]$, in which $\mathcal{B}_4$ honestly follows the protocol specification):

```
$\mathcal{B}_4(1^\lambda)$
$Q \leftarrow \emptyset$; $(pk_{\mathsf{LS}}, sk_{\mathsf{LS}}) \xleftarrow{r} \mathsf{KeyGen}(1^\lambda)$
$(\mathsf{snap}_1, \mathsf{snap}_2, (\mathsf{event}, \mathsf{rcpt}), \pi) \xleftarrow{r} \mathcal{A}^{\textsc{Msg}}(1^\lambda, pk_{\mathsf{LS}})$
return $(\mathsf{snap}_2[c], Q[c_2], \mathsf{event}, \pi)$

$\mathtt{CheckEntry}[\mathsf{LS}, 2, j](1^\lambda, \mathsf{state}_{\mathsf{LS}}, m)$
$(\mathsf{event}, \mathsf{snap}_{\mathsf{Au}}) \leftarrow m$
$\pi' \leftarrow \mathsf{ProveIncl}(c_{\mathsf{Au}}, \mathsf{event}, \mathsf{events})$
$Q[c_{\mathsf{LS}}] = \mathsf{events}$
return $(\mathsf{state}_{\mathsf{LS}}, \pi', \mathsf{Auditor}, \varepsilon)$
```

It is clear that the interaction with $\mathcal{B}_4$ is identical to the interaction that $\mathcal{A}$ expects, as $\mathcal{B}_4$ executes all the algorithms honestly. If E_4 holds, then it must be the case that $\mathsf{CheckNotIncl}(c_2, \mathsf{event}, \pi) = 1$. Because $(c_2, t_2) \in M_{\mathsf{snap}}$, $c_2 \in Q$, so a similar argument to that in the proof of Equation 3 shows that $\mathsf{CheckCom}(c_2, Q[c_2]) = 1$. Furthermore, because $\mathcal{B}_4$ behaves honestly we know that $Q[c_2]$ is ordered and that if $E_{\text{gd-event}}$ holds and if $\mathsf{time}(\mathsf{event}) \leq t_2$ then $\mathsf{event} \in \mathsf{events}$. Thus $\mathcal{B}_4$ wins whenever E_4 occurs.

Accountability. Let $\mathcal{A}$ be an adversary playing $\mathsf{G}^{\text{trace}}_{\mathcal{A}}(\lambda)$. We build an adversary $\mathcal{B}$ such that

$$\mathbf{Adv}^{\text{trace}}_{\text{trans},\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{\text{p-cons}}_{\text{dlc},\mathcal{B}}(\lambda),$$

from which the theorem follows.

To establish this, suppose there exists an event $\mathsf{event} \in (\mathsf{events}_{\mathsf{pledged}} \cap \mathsf{events}_{\mathsf{fail}}) \setminus \mathsf{events}_{\mathsf{Mo}}$. By the definition of the honest auditor and monitor, they output valid evidence $\mathsf{evidence} = (\mathsf{sth}_{\mathsf{Au}}, \mathsf{sth}_{\mathsf{Mo}}, (\mathsf{event}, \mathsf{rcpt}), \pi = \mathsf{DemoNotIncl}(c_{\mathsf{Mo}}, \mathsf{certs}_{\mathsf{Mo}}, \mathsf{event}))$ only if $\mathsf{CheckNotIncl}(c_{\mathsf{Mo}}, \mathsf{event}, \pi) = 1$.

Thus, if $\mathsf{CheckEvidence}(pk_{\mathsf{LS}}, \mathsf{evidence}) = 0$, it must be the case that

$$\mathsf{CheckNotIncl}(c_{\mathsf{Mo}}, \mathsf{event}, \mathsf{DemoNotIncl}(\mathsf{certs}_{\mathsf{Mo}}, \mathsf{event})) = 0,$$

in which case we can construct an adversary $\mathcal{B}$ against the provable non-inclusion of the DLC as follows (we omit the descriptions of the oracles, in which $\mathcal{B}$ executes the honest algorithm):

```
$\underline{\mathcal{B}(1^\lambda)}$
$\mathsf{events} \leftarrow \emptyset$; $a \leftarrow \emptyset$
$pk_{\mathsf{LS}} \xleftarrow{r} \mathcal{A}^{\text{Msg}}(1^\lambda)$
find $\mathsf{event} \in (\mathsf{events}_{\mathsf{pledged}} \cap \mathsf{events}_{\mathsf{fail}})$
return $(\mathsf{state}_{\mathsf{Mo}}[\mathsf{events}], \mathsf{event})$
```

By the honest behavior of the monitor, $\mathsf{isOrdered}(\mathsf{certs}_{\mathsf{Mo}}) = 1$, so the final winning condition of $\mathsf{G}^{\text{p-cons}}_{\mathcal{B}}(\lambda)$ is satisfied, and by assumption so is the first winning condition. It therefore remains to show that $\mathsf{event} \notin \mathsf{events}_{\mathsf{Mo}}$; this, however, follows by assumption as well. Thus, $\mathcal{B}$ succeeds whenever $\mathcal{A}$ does. $\square$