Redactable Blockchain
— or —
Rewriting History in Bitcoin and Friends

Giuseppe Ateniese[1], Bernardo Magri[2], Daniele Venturi[3], and Ewerton Andrade[4]

[1] *Stevens Institute of Technology, USA*
[2] *Sapienza University of Rome, Italy*
[3] *University of Trento, Italy*
[4] *University of São Paulo, Brazil*

August 5, 2016

Abstract

We put forward a new framework that makes it possible to re-write and/or compress the content of any number of blocks in decentralized services exploiting the blockchain technology. As we argue, there are several reasons to prefer an editable blockchain, spanning from the necessity to remove improper content and the possibility to support applications requiring re-writable storage, to "the right to be forgotten".

Our approach generically leverages so-called chameleon hash functions (Krawczyk and Rabin, NDSS '00), which allow to efficiently determine hash collisions given a secret trapdoor information. We detail how to integrate a chameleon hash function in virtually any blockchain-based technology, for both cases where the power of redacting the blockchain content is in the hands of a single trusted entity and where such a capability is distributed among several distrustful parties (as is the case in Bitcoin).

We also report on a proof-of-concept implementation of a redactable blockchain, building on top of Nakamoto's Bitcoin core. The implementation only requires minimal changes to the way current client software interprets information stored in the blockchain and to the current blockchain, block, or transaction structures. Moreover, our experiments show that the overhead imposed by a redactable blockchain is small compared to the case of an immutable one.

Keywords: Blockchain, Bitcoin, chameleon hash functions.

Contents

1 Introduction

The cost of the bankruptcy of Lehman Brothers in 2008 to the United States is estimated in trillions [U.S13] and triggered a chain of events that sent several countries into economic recession or depression. One contributor to the crisis was the centralized payment and monetary system based on clearinghouses that act as intermediaries between buyers and sellers and take on the risk of defaults. Unfortunately, clearinghouses add a significant cost to any interbank transactions and do not always operate transparently.

Bitcoin is an innovative technology that may allow banks to settle accounts between themselves without relying on centralized entities. It is considered the first decentralized currency system that works on a global scale. It relies on cryptographic proofs of work, digital signatures, and peer-to-peer networking to provide a distributed ledger (called the blockchain) containing transactions. Digital currency is, however, the simplest application of the blockchain technology. Bitcoin includes a scripting language that can be used to build more expressive "smart contracts," basically cryptographically-locked boxes that can be opened if certain conditions are verified. In addition, transactions can store arbitrary data via the OP_RETURN mechanism.

The blockchain technology promises to revolutionize the way we conduct business. Blockchain startups have received more than $1bn [Coi] of venture capital money to exploit this technology for applications such as voting, record keeping, contracts, etc. Conventional services are centralized and do not scale well. The blockchain allows services to be completely decentralized. There is no need to rely on, or trust, a single organization. It is a disruptive technology that will change the way money, assets and securities are currently managed. Business agreements can be encoded as smart contracts which in turn can handle automatically their executions along with the arbitration of disputes, thus reducing cost and providing more transparency. From a technology point of view, the blockchain is equally revolutionary. It provides for the first time a probabilistic solution to the Byzantine generals problem, where consensus is reached over time (after confirmations), and makes use of economic incentives to secure the overall infrastructure.

Two approaches have emerged to facilitate the use of the blockchain technology to implement decentralized services and applications (what is referred to as Bitcoin 2.0). The first "overlay" approach is to rely on the existing Bitcoin blockchain and build a new framework on top of it. This is done through transactions with OP_RETURN outputs which are unspendable and do not need to be stored in the UTXO database. The rationale of this approach is that the

Bitcoin blockchain already exists and is adopted by many, which makes it inherently more secure and resilient. However, certain constraints and constants set by the creator of Bitcoin (Satoshi) impede some (but not all) applications. For instance, blocks are mined every 10 minutes on average and the Bitcoin scripting language is not Turing-complete. This works perfectly for the currency, but forces other applications to get around these limitations through cumbersome hacks. The second approach is to build an alternative blockchain with all the desired features. This approach is gathering momentum (see, e.g, Ethereum [Eth]), and promises full decentralization. It enables very expressive smart contracts that achieve a high degree of automation.

1.1 Motivation

The append-only nature of the blockchain is essential to the security of the Bitcoin ecosystem. Transactions are stored in the ledger forever and are immutable. This fits perfectly with the currency system. However, we argue that an immutable ledger is not appropriate for all new applications that are being envisaged for the blockchain. Whether the blockchain is used to store data or code (smart contracts), there must be a way to redact its content in specific and exceptional circumstances. Redactions should be performed only under strict constraints, and with full transparency and accountability. Some examples where a redactable blockchain is desirable are outlined below.

(i) The ability to store arbitrary messages has already been abused, and now the Bitcoin blockchain contains child pornography, improper content, and material that infringes on intellectual rights (see e.g., [Hop, Pea, HC]). The intent of these abuses is to disrupt the Bitcoin system, since users may not be willing to participate and download the blockchain for fear of being prosecuted for possession of illegal or improper content on their computers. There are currently only 8-10K full nodes that store the entire blockchain and if this number declines, the Bitcoin ecosystem may be severely disrupted. In addition, improper content (gossip, pictures, etc.) may affect the life of people forever if it is not removed from the blockchain. Thus, appending new information is not an option in these cases.

(ii) Bitcoin 2.0 applications require re-writable storage. Smart contracts and overlay applications may not work or scale if the blockchain is immutable. A smart contract is essentially a sequence of instructions that a miner is going to run in exchange for a compensation. Amending a contract or patching code, by appending a new version of it, does not scale and wastes precious resources.

(iii) Is our society ready for permanent storage or perfect accountability? We believe it is not and indeed much effort is spent to promote the "right to be forgotten." New blockchain applications promise to store files, notarize documents, manage health records, coordinate IoT devices, administer assets, etc. But records should be expunged in case they contain errors or sensitive information, or when it is required by law. Even encryption may not help as keys are notoriously difficult to manage and are often leaked.

(iv) Several financial institutions are exploring the benefits of blockchain-based solutions to reduce cost and increase trust in interbank interactions. Budgets, transactions, and financial results are routinely consolidated to create meaningful reports while allowing entities to maintain distinct accounting structures. Consolidation is difficult to achieve with immutable blockchains, since it is impossible to consolidate past transactions without affecting any subsequent blocks.

1.2 Our Contributions

We propose an approach to make the blockchain redactable; by redaction we mean one of the following actions (and any combination of those): re-writing one or more blocks, compressing any

number of blocks into a smaller number of blocks, and inserting one or more blocks. Redactions can be made only by authorized entities and under specific constraints; moreover redactions are publicly auditable by existing miners, since they must approve the new blockchain and have access to its old copies. However, new miners are *oblivious*, given that the blockchain in our design is implemented as a history-independent data structure in the sense introduced by Naor and Teague [NT01]. That is, no information can be deduced about the past from the current view of the blockchain (also called *anti-persistence* in [NT01]).

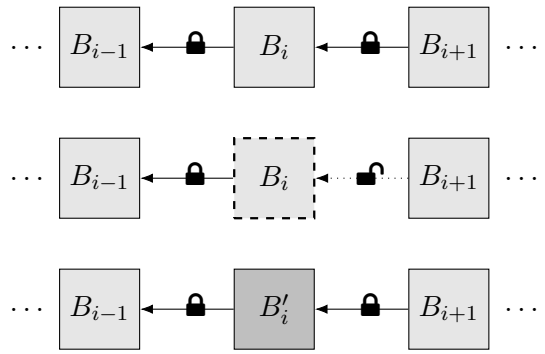


Figure 1: Redaction operations on a redactable blockchain. In the top blockchain, all padlocks are locked resulting in an immutable blockchain. In the middle blockchain, the padlock from block B_{i+1} to block B_i is open, meaning that the content of block B_i can be redacted. In the bottom blockchain, the block B_i was redacted (resulting in block B'_i) and all the padlocks are once again locked, making the blockchain immutable.

All blockchain designs rely on a hash chain that connects each block to the previous one, to create an immutable sequence. The immutability comes from the collision resistance property of the hash function. The best way to grasp the concept of a redactable blockchain is to think of adding a lock to each link of the hash chain (see Figure 1): Without the lock key it is hard to find collisions and the chain is immutable, but given the lock key it is possible to efficiently find collisions and thus replace the content of any block in the chain. With the knowledge of the key, any reduction is then possible: deletion, modification, and insertion of any number of blocks. Note that if the lock key is lost or destroyed, then a redactable blockchain reverts to an immutable one.

The main idea of our design is to employ a special hash function that is collision-resistant unless a trapdoor is known. This special hash is an evolution of a standard chameleon hash. Indeed, in a standard chameleon hash, collisions must be kept private since the trapdoor can be extracted from a single collision. In our improved design, it is safe to reveal any number of collisions.

Our contributions include:

- A new design for a redactable blockchain which is compatible with all popular blockchain proposals (cf. Section 3). Our blockchain is history-independent in the sense introduced by Naor and Teague [NT01]. The main feature of our system is that it is compatible with current blockchain designs, i.e., it can be implemented right now and requires only minimal changes to the way current client software interprets information stored in the blockchain (more on this later), and to the current blockchain, block, or transaction structures. We believe compatibility is an important feature that must be preserved.
- Improved chameleon hash design (cf. Section 4). Traditional chameleon hashes have the key exposure problem as observed in [AdM04], except the scheme proposed in [AdM04], but which relies on the generic group model. It was left open to find similarly enhanced chameleon hashes in the standard model. We first generalize the definition of chameleon hash to make it more relevant in practice, and then provide new constructions in the standard model through a generic transformation.
- Implementation (cf. Section 5). We developed a redactable blockchain prototype on top

of the Bitcoin core. We ran experiments where several blocks were modified or removed from a Bitcoin blockchain, thus showing the feasibility of our approach.

1.3 Remarks

Proposing to affect the immutability of the blockchain may seem an ill-conceived concept given the importance of the append-only nature of the blockchain. However, hard forks exist that can be used to undo recent transactions. As for hard forks, we expect redactions to occur in rare and exceptional circumstances.

Why not applying a hard fork in the past? Hard forks can be seen as the Undo operation and thus they make sense only for recently mined blocks. Imagine making a hard fork for a block added to the blockchain, say, 5 years ago. All subsequent blocks will be rendered invalid and all transactions from 5 years ago till now will have to be reprocessed. Thus, regenerating the blockchain will take another 5 years assuming similar mining power.

Would this redaction mechanism make sense for Bitcoin? We target Bitcoin 2.0 applications but we believe Bitcoin can also benefit from our solution. Consider this trivial but effective attack against Bitcoin. (i) Divide objectionable content (e.g., child or revenge pornography, sensitive or private information, etc.) in packets as it is done with TCP/IP. (ii) Store each packet within the OP_RETURN field of several Bitcoin transactions. (iii) After several blocks are mined, release a simple script or provide a web page where the improper content can be reconstructed as with TCP/IP packets. (iv) Wait for a lawsuit to be filed. If (when) this happens, then Bitcoin could be legally shut down and the blockchain removed for good. Notice that access to content on the Internet can be controlled, filtered out, or made it hard to find. On the other end, content in the blockchain must always be available and stored locally at each node.

Who can make redactions? We show how to make redactions given the knowledge of a secret key. This key could be in the hands of miners, a centralized auditor, or shares of the key could be distributed among several authorities. The actual way the trapdoor key is managed depends upon the requirements of specific applications; while we provide some examples (see Section 3.5), we stress that those are just a few possibilities out of many.

Why can't the blockchain be edited "by fiat", relying on meta-transactions? It is possible to create a block revocation list that miners are instructed to check and avoid. The problem however is that old blocks will still be there with the information that was supposed to be redacted. Thus, this approach is pointless. Another variant is to actually remove blocks, creating "holes" in the blockchain, and instruct miners to ignore those blocks. This approach is even worse since the blockchain is not valid anymore and exceptions must be hardcoded in the software of each miner or made available as an authenticated blacklist.

Couldn't the set of miners "vote" by their power, "sign" the new block and insert it into the correct position? No, because this is essentially a hard fork and all subsequent blocks will be invalid. Punching the blockchain makes it invalid and can only be handled as described in the previous point.

If trusted authorities can redact the blockchain, can't you get rid of PoW-based consensus? Redactions, as hard forks, are supposed to happen very rarely, in case of emergencies (e.g., the DAO attack) or when sensitive information is leaked (e.g., revenge porn). Editors do not operate daily but only in exceptional circumstances. They do not have the ability to run or maintain a blockchain. Trusted authorities could be individuals, such as judges or arbitrators, or/and organizations, such as the International Monetary Fund (IMF), the World Trade Organization (WTO), Electronic Frontier Foundation (EFF), INTERPOL, etc. They are not meant to operate blockchain infrastructures (as a simple analogy, consider that the Securities and Exchange Commission (SEC) is not meant to run stock and options exchanges or electronic securities markets but to intervene to enforce federal securities laws). Thus, we must rely on PoW-based consensus to run the blockchain.

Are these trusted authorities the same as in permissioned blockchain? Not necessarily. In permissioned blockchain only specified actors (banks, financial operators, individuals, etc.) can participate and post transactions. Some or all of these actors could be allowed to redact the blockchain if they collaborate. Shares of the chameleon hash key could be distributed among them so that the key is reconstructed when shares are pooled together according to some access control structure.

Could redactions have helped with the DAO attack? The DAO attack was resolved with a hard fork. Technically there is no substantial difference between hard forks and redactions for recent events. Our solution would help in case frauds or unintended errors are discovered much later, when it is too late to apply a hard fork and efficiently rebuild the blockchain.

Is blockchain immutability a chimera? The aftermath of the DAO attack shows that immutability is contentious (DAO is dead, lawsuits are looming, two parallel chains ETH/ETC were created, the future of Ethereum is in question, etc.). Other than affecting privacy (see [Ten]), immutability also affects scalability (see [DeR]). Our primary intent is to provide a technical answer to the question: "How can I make a redactable blockchain?". However, we do believe immutability of the blockchain should be reconsidered if Bitcoin 2.0 applications are to be turned from lab experiments to real deployments.

1.4 Related Work

Several papers have analyzed the properties and extended the features of the Bitcoin protocol (see, e.g., [ADMM14b, AFMdM14, ADMM15, PS15]). Bitcoin has also found several innovative applications far beyond its initial scope, e.g., to achieve fairness in secure multi-party computation [ADMM14c, ADMM14a, BK14], to build smart contracts [KMS+15, BDM16], to distributed cryptography [AD15], and more [KMB15, KT15, RKS15]. Blockchain based technologies, and the properties they achieve, were also studied in recent work, both for the synchronous [GKL15] and asynchronous [PSas16] network model.

1.5 Roadmap

We start by introducing some standard notation in Section 2. Then, we explain how to integrate chameleon hash functions into current blockchain-based technologies with the purpose of creating a redactable blockchain, in Section 3. Section 4 contains the details of our generic transformation yielding chameleon hash functions with key-exposure freeness. In Section 5, we comment on a proof-of-concept implementation deploying our framework within Bitcoin. We finally give some directions for future research, in Section 6.

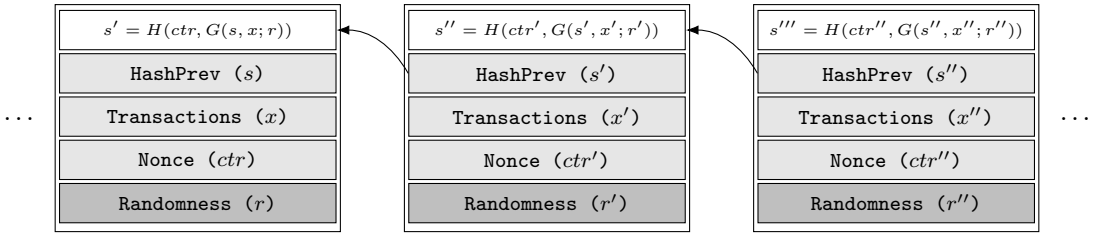


Figure 2: The redactable blockchain structure (using a public-coin chameleon hash). The field s of a block stores the value shown in the top white field of the previous block. We note that the top white field is not stored in the block. The bottom darker field (`Randomness`) is updated when the block is redacted (i.e., a collision is computed).

2 Notation

For a string x, we denote its length by $|x|$; if $\mathcal{X}$ is a set, $|\mathcal{X}|$ represents the number of elements in $\mathcal{X}$. When x is chosen randomly in $\mathcal{X}$, we write $x \leftarrow\!\!\$\, \mathcal{X}$. When A is an algorithm, we write $y \leftarrow\!\!\$\, \mathsf{A}(x)$ to denote a run of A on input x and output y; if A is randomized, then y is a random variable and $\mathsf{A}(x;r)$ denotes a run of A on input x and randomness r. An algorithm A is *probabilistic polynomial-time* (PPT) if A is randomized and for any input $x, r \in \{0,1\}^*$ the computation of $\mathsf{A}(x;r)$ terminates in at most $poly(|x|)$ steps.

We denote with $\kappa \in \mathbb{N}$ the security parameter. A function $\nu : \mathbb{N} \to [0,1]$ is negligible in the security parameter (or simply negligible) if it vanishes faster than the inverse of any polynomial in κ, i.e. $\nu(\kappa) = \kappa^{-\omega(1)}$.

For a random variable $\mathbf{X}$, we write $\mathbb{P}[\mathbf{X} = x]$ for the probability that $\mathbf{X}$ takes on a particular value $x \in \mathcal{X}$ (where $\mathcal{X}$ is the set where $\mathbf{X}$ is defined). Given two ensembles $\mathbf{X} = \{X_\kappa\}_{\kappa\in\mathbb{N}}$ and $\mathbf{Y} = \{Y_\kappa\}_{\kappa\in\mathbb{N}}$, we write $\mathbf{X} \equiv \mathbf{Y}$ to denote that the two ensembles are identically distributed, and $\mathbf{X} \approx_c \mathbf{Y}$ to denote that they are computationally indistinguishable.

3 Redacting the Blockchain

In this section we introduce our framework, explaining how to modify current blockchain technologies in order to obtain a redactable blockchain. We start with a brief description of a blockchain abstraction, due to Garay, Kiayias and Leonardos [GKL15], in Section 3.1. In Section 3.2 we recall the concept of chameleon hash functions. We then put forward two new algorithms that can be used to re-write the content of the blockchain, both in the centralized setting where a trusted party is in charge of rewriting the blocks (in Section 3.3) and in the decentralized setting where no such trusted party is available (in Section 3.4). Finally, in Section 3.5, we comment on how the chameleon hash keys can be managed in a few concrete scenarios.

3.1 Blockchain Basics

We make use of the notation of [GKL15] to describe the blockchain. A block is a triple of the form $B = \langle s, x, ctr\rangle$, where $s \in \{0,1\}^\kappa$, $x \in \{0,1\}^*$ and $ctr \in \mathbb{N}$. Block B is *valid* if

$$\mathbf{validblock}_q^D(B) := (H(ctr, G(s,x)) < D) \wedge (ctr < q) = 1.$$

Here, $H : \{0,1\}^* \to \{0,1\}^\kappa$ and $G : \{0,1\}^* \to \{0,1\}^\kappa$ are collision-resistant hash functions, and the parameters $D \in \mathbb{N}$ and $q \in \mathbb{N}$ are the block's difficulty level and the maximum number of hash queries that a user is allowed to make in any given round of the protocol, respectively.

The *blockchain* is simply a chain (or sequence) of blocks, that we call $\mathcal{C}$. The rightmost block is called the head of the chain, denoted by $\mathsf{Head}(\mathcal{C})$. Any chain $\mathcal{C}$ with a head $\mathsf{Head}(\mathcal{C}) := \langle s, x, ctr\rangle$ can be extended to a new longer chain $\mathcal{C}' := \mathcal{C}||B'$ by attaching a (valid) block $B' := \langle s', x', ctr'\rangle$ such that $s' = H(ctr, G(s, x))$; the head of the new chain $\mathcal{C}'$ is $\mathsf{Head}(\mathcal{C}') = B'$. A chain $\mathcal{C}$ can also be empty, and in such a case we let $\mathcal{C} = \varepsilon$. The function $\mathsf{len}(\mathcal{C})$ denotes the length of a chain $\mathcal{C}$ (i.e., its number of blocks). For a chain $\mathcal{C}$ of length n and any $k \geq 0$, we denote by $\mathcal{C}^{\lceil k}$ the chain resulting from removing the k rightmost blocks of $\mathcal{C}$, and analogously we denote by ${}^{k\rceil}\mathcal{C}$ the chain resulting in removing the k *leftmost* blocks of $\mathcal{C}$; note that if $k \geq n$ then $\mathcal{C}^{\lceil k} = \varepsilon$ and ${}^{k\rceil}\mathcal{C} = \varepsilon$. If $\mathcal{C}$ is a prefix of $\mathcal{C}'$ we write $\mathcal{C} \prec \mathcal{C}'$. We also note that the difficulty level D can be different among blocks in a chain.

The work of [GKL15] models the Bitcoin protocol in a setting where the number of participants is always fixed and the network in synchronized. They show that the protocol satisfies *consistency* in this model, meaning that all honest participants have the same chain prefix of the blockchain. A more recent work by Pass, Seeman and shelat [PSas16] analyses the case where the network is asynchronous and the number of participants can dynamically change. We point that our framework is independent of the network type in these models.

3.2 Chameleon Hash Functions

The concept of chameleon hashing was put forward by Krawczyk and Rabin [KR00], building on the notion of chameleon commitments [BCC88]. Informally, a chameleon hash is a cryptographic hash function that contains a trapdoor: Without the trapdoor it should be hard to find collisions, but knowledge of the trapdoor information allows to efficiently generate collisions for the hash function.

Secret-coin hashing. We start by introducing a generalization of the standard concept of chameleon hashing to make it more relevant in practice. Our generalization is referred to as "secret-coin" and includes standard chameleon hashes as a special case (now referred to as "public-coin").

Definition 1 (Secret-coin chameleon hash)**.** A *secret-coin* chameleon hash function is a tuple of efficient algorithms $\mathcal{CH} = (\mathsf{HGen}, \mathsf{Hash}, \mathsf{HVer}, \mathsf{HCol})$ specified as follows.

- $(hk, tk) \leftarrow\!\!\$\ \mathsf{HGen}(1^\kappa)$: The probabilistic key generation algorithm HGen takes as input the security parameter $\kappa \in \mathbb{N}$, and outputs a public hash key hk and a secret trapdoor key tk.
- $(h, \xi) \leftarrow\!\!\$\ \mathsf{Hash}(hk, m)$: The probabilistic hashing algorithm Hash takes as input the hash key hk, a message $m \in \mathcal{M}$, and implicit random coins $r \in \mathcal{R}_{\text{hash}}$, and outputs a pair (h, ξ) that consists of the hash value h and a check string ξ.
- $d = \mathsf{HVer}(hk, m, (h, \xi))$: The deterministic verification algorithm HVer takes as input a message $m \in \mathcal{M}$, a candidate hash value h, and a check string ξ, and returns a bit d that equals 1 if (h, ξ) is a valid hash/check pair for the message m (otherwise d equals 0).
- $\pi' \leftarrow\!\!\$\ \mathsf{HCol}(tk, (h, m, \xi), m')$: The probabilistic collision finding algorithm HCol takes as input the trapdoor key tk, a valid tuple (h, m, ξ), and a new message $m' \in \mathcal{M}$, and returns a new check string ξ' such that $\mathsf{HVer}(hk, m, (h, \xi)) = \mathsf{HVer}(hk, m', (h, \xi')) = 1$. If (h, ξ) is not a valid hash/check pair for message m then the algorithm returns $\perp$.

Correctness informally says that a pair (h, ξ), computed by running the hashing algorithm, verifies with overwhelming probability.

Definition 2 (Correctness for chameleon hashing)**.** Let $\mathcal{CH} = (\mathsf{HGen}, \mathsf{Hash}, \mathsf{HVer}, \mathsf{HCol})$ be a secret-coin chameleon hash function with message space $\mathcal{M}$. We say that $\mathcal{CH}$ satisfies correctness

if for all $m \in \mathcal{M}$ there exists a negligible function $\nu : \mathbb{N} \to [0,1]$ such that

$$\mathbb{P}\left[\mathsf{HVer}(hk, m, (h,\xi)) = 1 : (h,\xi) \leftarrow\!\!\$\ \mathsf{Hash}(hk, m); (hk, tk) \leftarrow\!\!\$\ \mathsf{HGen}(1^\kappa)\right] \geq 1 - \nu(\kappa).$$

Public-coin hashing. In the definition above the hashing algorithm is randomized, and, upon input some message m, it produces a hash value h together with a check value ξ that helps verifying the correct computation of the hash given the public hash key. The random coins of the hashing algorithm are, however, secret. A particular case is the one where the check value ξ consists of the random coins r used to generate h, as the hash computation becomes completely deterministic once m and r are fixed; we call such a chameleon hash function *public-coin* and we define it formally below. Since the verification algorithm simply re-runs the hashing algorithm, we typically drop the verification algorithm from $\mathcal{CH}$ in the case of public-coin chameleon hashing.

Definition 3 (Public-coin chameleon hash)**.** A *public-coin* chameleon hash is a collection of efficient algorithms $\mathcal{CH} = (\mathsf{HGen}, \mathsf{Hash}, \mathsf{HVer}, \mathsf{HCol})$ specified as in Definition 1, with the following differences:

- The hashing algorithm Hash, upon input the hash key hk and message $m \in \mathcal{M}$, returns a pair (h, r), where $r \in \mathcal{R}_{\text{hash}}$ denote the implicit random coins used to generate the hash value.
- The verification algorithm HVer, given as input the hash key hk, message m, and a pair (h, r), returns 1 if and only if $\mathsf{Hash}(hk, m; r) = h$.

Collision resistance. The main security property satisfied by a secret/public-coin chameleon hash function is that of collision resistance: No PPT algorithm, given the public hash key hk, can find two pairs (m, ξ) and (m', ξ') that are valid under hk and such that $m \neq m'$, with all but a negligible probability. Furthermore, for the applications we devise in this paper, it is important that the above still holds even after seeing arbitrary collisions generated using the trapdoor key tk corresponding to hk. We refer the reader to Section 4 for formal security definitions, and for a generic construction achieving such a strong form of security.

3.3 Centralized Setting

The main idea behind our approach is to set the inner hash function (i.e., the function G), used to chain the different blocks in the blockchain, to be a chameleon hash function. Intuitively, re-writing the content of each block is possible by finding collisions in the hash function (without modifying the outer hash function H). Below, we detail this idea in the simple setting where only a single (trusted) central authority is able to redact the blockchain; see Section 3.5 for concrete examples where this case applies.

In order for the above to work, we require some modifications to the previously defined block. A block is now a tuple $B := \langle s, x, ctr, (h, \xi)\rangle$, where the components s, x and ctr are the same as before, and the new component (h, ξ) is the hash/check pair for a chameleon hash. The function G is defined to be a secret-coin chameleon hash $\mathcal{CH} = (\mathsf{HGen}, \mathsf{Hash}, \mathsf{HVer}, \mathsf{HCol})$, and the validation predicate for a block is now equal to

$$\mathbf{validblock}_q^D(B) := (H(ctr, h) < D) \wedge (\mathsf{HVer}(hk, (s, x), (h, \xi))) \wedge (ctr < q) = 1.$$

Given a chain $\mathcal{C}$ with head $\mathsf{Head}(\mathcal{C}) := \langle s, x, ctr, (h, \xi)\rangle$, we can extend it to a longer chain by attaching a (valid) block $B' := \langle s', x', ctr', (h', \xi')\rangle$ such that $s' = H(ctr, h)$.

Notice that the domain of the chameleon hash can be easily adjusted to the proper size by first hashing the input of Hash with a regular collision-resistant hash of the desired output size. We also stress that the verification of a chameleon hash value needs to be computed by its own verification function (i.e., by running HVer), and not simply by recomputing the hash, like it is done with standard (deterministic) hash functions.

The case where the chameleon hash is public-coin can be cast as a special case of the above. However, note that there is no need for storing the hash value h, as this value can be computed as a deterministic function of the chameleon hash function's input and randomness. Thus, in this case, a block has a type $B := \langle s, x, ctr, r\rangle$, where r is the randomness for the chameleon hash. The validation predicate for a block becomes

$$\mathbf{validblock}_q^D(B) := (H(ctr, \mathsf{Hash}(hk, (s, x); r)) < D) \wedge (ctr < q) = 1.$$

Finally, given a chain $\mathcal{C}$ with head $\mathsf{Head}(\mathcal{C}) := \langle s, x, ctr, r\rangle$, we can extend it to a longer chain by attaching a (valid) block $B' := \langle s', x', ctr', r'\rangle$ such that $s' = H(ctr, \mathsf{Hash}(hk, (s, x); r))$. See Fig. 2 for a pictorial representation.

Rewriting blocks. Next, we define a chain redacting algorithm (see Algorithm 1 below) that takes as input a chain $\mathcal{C}$ to be redacted, a set of indices that represents the positions (in the chain $\mathcal{C}$) of the blocks that are going to be redacted, and another set with the new x''s values for each of the blocks to be redacted. The algorithm also takes as input the chameleon hash trapdoor key tk. The intuition behind it is that, for each block to be redacted, we compute a collision for the hash of the block with its new content x'. A new chain $\mathcal{C}'$ is created by replacing the original block with its modified counterpart. We note that at the end of the execution of Algorithm 1, the central authority should broadcast the new redacted chain as a *special* chain, meaning that every user of the system should adopt this new redacted chain in favor of any other chain, even longer ones. The way this is achieved depends on the actual system in use.

Algorithm 1: Chain Redact

input : The input chain $\mathcal{C}$ of length n, a set of block indices $\mathcal{I} \subseteq [n]$, a set of values $\{x'_i\}_{i\in\mathcal{I}}$, and the chameleon hash trapdoor key tk.
output: The redacted chain $\mathcal{C}'$ of length n.

$\mathcal{C}' \leftarrow \mathcal{C}$;
Parse the chain $\mathcal{C}'$ as $(B_1, \cdots, B_n)$;
for $i := 1, \ldots, n$ **do**
  if $i \in \mathcal{I}$ **then**
    Parse the i-th block of $\mathcal{C}'$ as $B_i := \langle s_i, x_i, ctr_i, (h_i, \xi_i)\rangle$;
    $\xi'_i \leftarrow \mathsf{HCol}(tk, (h_i, s_i||x_i, \xi_i), (s_i||x'_i))$;
    $B'_i := \langle s_i, x'_i, ctr_i, (h_i, \xi'_i)\rangle$;
    $\mathcal{C}' \leftarrow \mathcal{C}'^{\lceil n-i+1}||B'_i||^{i\rceil}\mathcal{C}'$;
  end
end
return $\mathcal{C}'$

Note that each time a block is redacted using Algorithm 1, a collision for the underlying chameleon hash function is exposed. Hence, it is important that the ability to see arbitrary collisions does not expose the secret trapdoor key, as otherwise unauthorized users might be able to rewrite arbitrary blocks in the chain. In Section 4 we explain how to generically leverage any standard collision-resistant chameleon hash function into one additionally meeting such a key-exposure freeness requirement.

Shrinking the chain. Another possibility with redactable blockchains is to completely *remove* entire blocks from a chain. This can be essential for scalability purposes, such as saving disk space and computational power necessary when handling larger chains. We present an algorithm (see Algorithm 2 below) for such a "chain shrinking" functionality. The intuition behind it is that in order to remove the block B_i it is necessary to redact the block B_{i+1} by assigning $s_{i+1} \leftarrow s_i$. A collision then needs to be computed for B_{i+1} producing the new block B'_{i+1} that is inserted in the chain in place of the B_{i+1} block, leaving the chain in a consistent state. As in Algorithm 1, we also note that at the end of the execution of Algorithm 2, the central authority should broadcast the new shrinked chain as a *special* chain, meaning that every user of the system should adopt this new redacted chain in favor of any other chain, even longer ones.

Algorithm 2: Chain Shrink

input : The input chain $\mathcal{C}$ of length n, a set of block indices $\mathcal{I} \subseteq [n]$ and the chameleon hash trapdoor key tk.
output: The new chain $\mathcal{C}'$ of length $n - |\mathcal{I}|$.

$\mathcal{C}' \leftarrow \mathcal{C}$;
Parse the chain $\mathcal{C}'$ as $(B_1, \cdots, B_n)$;
for $i := 1, \ldots, n$ **do**
  if $i \in \mathcal{I}$ **then**
    Parse the i-th block of $\mathcal{C}'$ as $B_i := \langle s_i, x_i, ctr_i, (h_i, \xi_i)\rangle$;
    Parse the $i+1$-th block of $\mathcal{C}'$ as $B_{i+1} := \langle s_{i+1}, x_{i+1}, ctr_{i+1}, (h_{i+1}, \xi_{i+1})\rangle$;
    $\xi'_{i+1} \leftarrow \mathsf{HCol}(tk, (h_{i+1}, s_{i+1}||x_{i+1}, \xi_{i+1}), (s_i||x_{i+1}))$;
    $B'_{i+1} := \langle s_i, x_{i+1}, ctr_{i+1}, (h_{i+1}, \xi'_{i+1})\rangle$;
    $\mathcal{C}' \leftarrow \mathcal{C}'^{\lceil n-i}||B'_{i+1}||^{i+1\rceil}\mathcal{C}'$;
  end
end
return $\mathcal{C}'$

We note that in Algorithm 2, if the set $\mathcal{I}$ contains only indexes to successive blocks, the execution can be optimized to essentially one execution of the *for* loop. This is because in order to remove blocks B_k to B_{k+j}, it is sufficient to redact only block B_{k+j+1} (i.e., the next remaining block).

3.4 Decentralized Setting

Below, we explain how to adapt our framework to the decentralized setting, where there is no central trusted authority. The main idea is to have the trapdoor key be secretly shared among some fixed set of users that are in charge of redacting the blockchain. When a block needs to be redacted, the users from this set engage in a secure multiparty computation (MPC) protocol to compute Algorithm 1 and Algorithm 2 in a fully distributed manner.

3.4.1 Ideal Functionalities

During the set up of the system, we fix a subset $\mathcal{U}$ of cardinality n, containing the users that will be in charge of redacting the blockchain content. We remark that the actual choice of the subset $\mathcal{U}$ can be completely dependent on the application and on the system requirements; we discuss some examples in Section 3.5.

Key Generation Functionality:

1. After receiving the "start" signal from all honest parties, run $(hk, tk) \leftarrow_{\$} \mathsf{HGen}(1^{\kappa})$ and send hk to the adversary.
2. We assume a secret sharing scheme $(\mathsf{Share}, \mathsf{Rec})$ is given, with which the trapdoor key tk can be secret-shared. For each dishonest party P_j, receive a share τ_j from the adversary.
3. Construct a complete set of shares $(\tau_1, \cdots, \tau_n)$ for the trapdoor key tk taking into consideration all the dishonest shares sent by the adversary. We note that it is always possible to construct such a set of shares since all the dishonest parties form an unqualified set for the secret sharing scheme. Send τ_i to each honest party P_i.

Figure 3: The ideal functionality for the distributed key generation

Following the common practice in the setting of MPC, we now define two ideal functionalities that aim at capturing the security requirements for generating the hash keys and for redacting the blockchain in the decentralized setting. These functionalities will later be realized by concrete MPC protocols, in both cases of semi-honest and fully malicious corruptions.

Key generation. When the system is set-up for the first time, we need to run the key generation algorithm HGen for the underlying chameleon hash function, obtaining a public hash key hk and a secret trapdoor key tk. Since no user is allowed to know the trapdoor key, the idea is to have each player P_i in the set $\mathcal{U}$ obtain a share τ_i of tk. This is the purpose of the ideal functionality described in Fig. 3, which is parametrized by a secret sharing scheme $(\mathsf{Share}, \mathsf{Rec})$.

Recall that a t-out-of-n secret sharing scheme $(\mathsf{Share}, \mathsf{Rec})$ consists of a pair of algorithms such that: (i) The randomized algorithm Share takes as input a target value x and returns a sequence of n shares $\tau_1, \ldots, \tau_n$; (ii) The deterministic algorithm Rec takes as input n shares $\tau_1, \ldots, \tau_n$ and returns a value x or an incorrect output symbol $\perp$. The main security guarantee is that any subset of t shares (a.k.a. an unqualified set) reveals no information on the shared value x; on the other hand, any subset of $t+1$ (or more) shares allows to efficiently recover x. We refer the reader, e.g., to [Bei11] for more details on secret sharing; some examples are also discussed below.

Chain redaction. When a block $B := \langle s, x, ctr, (h, \xi) \rangle$ needs to be redacted into a modified block $B' := \langle s, x', ctr, (h, \xi') \rangle$, each user in the set $\mathcal{U}$ needs to inspect its own blockchain and find block B. Hence, the players need to execute Algorithm 1 in a distributed manner. In particular, each player P_i is given as input its own share τ_i of the chameleon hash trapdoor key, and they all need to run the collision-finding algorithm HCol on common input $((h, s||x, \xi), s||x')$, in order to obtain the modified check value ξ'.

This is the purpose of the ideal functionality described in Fig. 4, which is again parametrized by a secret sharing scheme $(\mathsf{Share}, \mathsf{Rec})$ (in fact, the same secret sharing scheme as for the functionality of Fig. 3). For simplicity, we described the functionality for the general case where the goal is to find collisions between arbitrary messages m and m'. Note that actively corrupted players might submit incorrect shares, and the secret sharing scheme needs to cope with such a possibility. Also note that after each player receives the modified value ξ' for the new block B', each of the users in $\mathcal{U}$ constructs a new chain by replacing block B with block B'. Thus, the redacted chain is broadcast to all users in the system as a new *special* chain that should replace any other chain, even longer ones. Although the latter needs to be done in an application-

Collision-Finding Functionality:

1. Receive the shares τ_i from each party P_i and reconstruct the trapdoor key $tk := \mathsf{Rec}(\tau_1, \cdots, \tau_n)$. Note that the shares of the dishonest parties are chosen by the adversary.
2. Upon receiving a "compute collision" signal for the pair $((h, m, \xi), m')$ from all honest parties, compute $\xi' \leftarrow \mathsf{HCol}(tk, (h, m, \xi), m')$ and send (h, m, ξ) and ξ' to the adversary.
3. Upon receiving an "OK" signal from the adversary forward the value ξ' to all honest parties, otherwise forward $\perp$ to all honest parties.

Figure 4: The ideal functionality for the distributed collision-finding algorithm

specific manner, we recall that in practice the redact operation is not going to be performed very often, but only in case there is a need to redact undesirable content from a given block.

The decentralized version of Algorithm 2 is similar to the one described above, the only difference being that instead of redacting a block B_i, a new chain is built without the block B_i in it. To keep the chain valid the block B_{i+1} needs to be redacted, as detailed in Algorithm 2. The latter can be achieved using the same ideal functionality as in Fig. 4, by simply adjusting the input messages from the users.

3.4.2 Concrete Instantiations

We now present concrete protocols for securely realizing the ideal functionalities described in the previous section. For the sake of concreteness and practicality, we chose to work with the (public-coin) chameleon hash function introduced by Ateniese and de Medeiros [AdM04]; this construction satisfies enhanced collision resistance (cf. Definition 4) in the generic group model, based on the Discrete Logarithm assumption. After presenting the hash function, we deal separately with the setting in which the corrupted players within the set $\mathcal{U}$ are assumed to be semi-honest (i.e., they always follow the protocol but try to learn additional information from the transcript) and fully malicious (i.e., they can arbitrarily deviate from the protocol description).

The hash function. Let p, q be prime such that $p = 2q + 1$, and let g be a generator for the subgroup of quadratic residues $\mathbb{QR}_p$ of $\mathbb{Z}_p^*$. Consider the following public-coin chameleon hash function $(\mathsf{HGen}, \mathsf{Hash}, \mathsf{HCol})$.

- $(y, x) \leftarrow_\$ \mathsf{HGen}(1^\kappa)$: The trapdoor key tk is a random value $x \in [1, q-1]$, and the hash key hk is equal to $y = g^x$.
- $h := \mathsf{Hash}(y, m; r, s)$: To hash a message $m \in \{0,1\}^*$, pick random $r, s \leftarrow_\$ \mathbb{Z}_q$, and return $h := r - (y^{H(m||r)} \cdot g^s \bmod p) \bmod q$ where $H : \{0,1\}^* \to \mathbb{Z}_q$ is a standard-collision resistant hash function.
- $(r', s') \leftarrow_\$ \mathsf{HCol}(x, (h, m, r, s), m')$: To compute a collision for message m', pick a random $k \in [1, q-1]$ and compute $r' := h + (g^k \bmod p) \bmod q$ and $s' := k - H(m'||r') \cdot x \bmod q$. Return (r', s').

Semi-honest setting. As a warm up we consider the case of passive corruption, where up to t players in the set $\mathcal{U}$ are semi-honest. For this setting, we will rely on the following simple secret sharing scheme $(\mathsf{Share}, \mathsf{Rec})$: (i) Upon input a value $x \in \mathbb{Z}_q$, algorithm Share samples random $\tau_1, \ldots \tau_{n-1} \leftarrow_\$ \mathbb{Z}_q$, sets $\tau_n := x - \sum_{i=1}^{n-1} \tau_i \bmod q$, and returns $(\tau_1, \ldots, \tau_n)$; (ii) Algorithm Rec

takes as input $\tau_1, \ldots, \tau_n \in \mathbb{Z}_q$ and returns $x = \sum_{i=1}^{n} \tau_i \bmod q$. The above is easily seen to be an $(n-1)$-out-of-n secret sharing scheme.

Next, we describe two simple MPC protocols Π^1_{sh} and Π^2_{sh} for securely realizing the functionality of Fig. 3 and Fig. 4 (respectively).

- Consider the following n-party protocol Π^1_{sh}. Each player P_i picks a random $\tau_i \in \mathbb{Z}_q$ and then all players engage into a semi-honest MPC protocol for computing $y = \prod_{i=1}^{n} g^{\tau_i} \bmod p$; each player outputs (y, τ_i). This protocol is easily seen to realize the functionality of Fig. 3 under semi-honest corruption of up to $n-1$ players. Indeed, as long as one of the players is honest, the value y (with corresponding trapdoor $x := \sum_{i=1}^{n} \tau_i$) will be uniformly distributed, as required.
- Consider the following n-party protocol Π^2_{sh}, on common input $((h, m, r, s), m')$. First, each player P_i chooses a random $k_i \leftarrow\!\!\$\, \mathbb{Z}_q$ and then all players engage into a semi-honest MPC protocol for computing $r' := h + \left(\prod_{i=1}^{n} g^{k_i} \bmod p\right) \bmod q$. Second, the players engage into a semi-honest MPC protocol for computing $\sum_{i=1}^{n} k_i - H(m'||r') \cdot \sum_{i=1}^{n} \tau_i \bmod q$, where the private input of P_i is defined to be (k_i, τ_i). Finally, each player outputs (r', s'). The above protocol can be easily seen to securely realize the functionality of Fig. 4 under semi-honest corruptions. The number of tolerated corruptions depends on the semi-honest MPC protocols for performing the computations described above. Suitable protocols, for the setting where at least half of the players are honest, are described, e.g., in [BGW88, AL11].

Malicious setting. We briefly explain how to extend the previous protocol to the setting of active corruptions. The main difficulty here is that malicious players can now use incorrect shares. In order to ensure that the correct trapdoor is re-constructed, we rely on so called *robust* secret sharing. Informally, a secret sharing scheme $(\mathsf{Share}, \mathsf{Rec})$ is δ-robust if an adversary adaptively modifying at most t shares computed via Share can cause the output of Rec to be wrong with probability at most δ. See, e.g., [RB07, BPRW15] for a formal definition.

Before adapting the protocols, we recall the standard secret sharing scheme due to Shamir [Sha79]: (i) Upon input a value $x \in \mathbb{Z}_q$, algorithm Share picks random coefficients $\alpha_1, \ldots, \alpha_{t-1} \in \mathbb{Z}_q$ and defines $\tau_i := x + \alpha_1 \cdot i + \cdots + \alpha_{t-1} \cdot i^{t-1} \bmod q$ for all $i \in [n]$; (ii) Upon input t shares $(\tau_1, \ldots, \tau_t)$, algorithm Rec interpolates the polynomial $\alpha(X) = \alpha_0 + \alpha_1 \cdot X + \cdots + \alpha_{t-1} \cdot X^{t-1}$ and returns $\alpha(0)$. Shamir's secret sharing can be made robust against corruption of at most $t < n/3$ shares when used in tandem with Reed-Solomon decoding during the reconstruction procedure. Alternatively, for tolerating a higher threshold $t < n/2$ (which is also the maximal threshold for such schemes [Man11]) one could use Shamir's secret sharing in conjunction with information-theoretic message authentication codes, as proposed by Rabin and Ben-Or [RB89]. This results in shares of sub-optimal size $\mu + \tilde{O}(n \cdot \kappa)$, where μ is the bit-size of the message, and κ is the security parameter ensuring $\delta = 2^{-\kappa}$; robust schemes with almost optimal share size have recently been designed in [BPRW15].

Below is a sketch of how the protocols Π^1_{sh} and Π^2_{sh} described above can be adapted to the malicious setting, where for simplicity we use Shamir's secret sharing in tandem with Reed-Solomon decoding.

- Consider the following n-party protocol Π^1_{mal}. Each player P_i samples uniformly at random $x_i := (\alpha^i_0, \alpha^i_1, \ldots, \alpha^i_{t-1}) \in \mathbb{Z}^t_q$. Hence, the players engage in an MPC protocol for computing the function $(x_1, \ldots, x_n) \mapsto ((y, \alpha(1)), \ldots, (y, \alpha(n)))$, where $y = g^{\alpha(0)}$ and

$$\alpha(X) := \sum_{i=1}^{n} \alpha^i_0 + \sum_{i=1}^{n} \alpha^i_1 \cdot X + \cdots + \sum_{i=1}^{n} \alpha^i_{t-1} \cdot X^{t-1}.$$

- Consider the following n-party protocol Π^2_{mal}. The protocol proceeds similarly to Π^2_{sh} with the following differences. In the first step the random value k is shared among the players using Shamir's secret sharing (as done in Π^1_{mal}); denote by $\beta(X)$ the corresponding polynomial, and by $\beta(i)$ the share obtained by player P_i. In the second step the players engage in an MPC protocol for computing the value $r' = h + (g^{\hat{\beta}(0)} \bmod p) \bmod q$, where the polynomial $\hat{\beta}(X)$ is reconstructed by using the (possibly corrupted) shares $\beta(i)$ from the players, via the Berlekamp-Welch [WB86] algorithm. In the third step the players engage in another MPC protocol for computing $s' = \hat{\beta}(0) - H(m'||r'') \cdot \hat{\alpha}(0) \bmod q$, where the private input of each player P_i is $(\alpha(i), \beta(i))$ and $\hat{\alpha}(X)$, $\hat{\beta}(X)$ are again reconstructed by using the (possibly corrupted) shares $\alpha(i)$, $\beta(i)$ from the players, via the Berlekamp-Welch algorithm.

Note that the above protocols rely on auxiliary MPC protocols with malicious security, for computing arithmetic functions in $\mathbb{Z}_q$. Suitable MPC protocols for the above tasks, for the setting where at least two thirds of the players are honest, are described, e.g., in [BGW88, AL11, DFK+06].

3.5 On Key Management

Although we view the technical tools that make redactions possible as the main contribution of this work, a natural question that may arise is how the trapdoor key for the chameleon hash function is managed. We stress that the answer to this question is completely application dependent, but we still provide some examples.

Below we briefly describe three types of blockchains that occur in real-world applications [But], and clarify how the trapdoor key could be managed in each case.

- **Private blockchain**: In this type of blockchain, which is widely used by the financial sector [PB], the write permissions are only given to a central authority, and the read permissions may be public or restricted. In this case the key management becomes simple; the trapdoor key could be given to the central authority that has the power to compute collisions and therefore redact blocks. This scenario is described in Section 3.3.
- **Consortium blockchain**: In this type of blockchain the consensus is controlled by a predetermined set of parties (i.e., a consortium). In this case the trapdoor key can be shared among all the parties of the consortium, and redactions can be realized using MPC, as described in Section 3.4.
- **Public blockchain**: This type of blockchain is completely decentralized, and any party is allowed to send transactions to the network and have them included in the blockchain (as long as the transactions are valid). The consensus process is decentralized and not controlled by any party. The best example of a public blockchain is Bitcoin. In this case we have two options to manage the trapdoor key (both using MPC, as described in Section 3.4).
  1. The trapdoor key can be distributed among all the parties (full miners) of the network. The drawback of this solution is that, if the number of parties in the network is too big (e.g., > 200), it might not be very efficient due to performance issues of the MPC protocol.
  2. The trapdoor key can be distributed among a carefully chosen subset of the parties. For example, in Bitcoin it is well known that the majority of the network hashing power is actually controlled by a small number of parties (e.g., the top 7 mining pools control almost 70% of the network total hashing power [Inf]). Although we acknowledge that the concentration of hashing power to a small number of parties

can be unhealthy to the system, this solution does not change the existing Bitcoin trust assumption (i.e., Bitcoin already assumes trusted majority).

4 Chameleon Hash Transform

We start by formally defining collision resistance of public/secret coin chameleon hash functions, in Section 4.1. Section 4.2 introduces the main ingredients required by our generic transformation, which is described and analyzed in full details in Section 4.3. Finally, in Section 4.4, we instantiate our transformation under standard complexity assumptions, both in the standard and in the random oracle model.

4.1 Enhanced Collision Resistance

A *collision* for a secret-coin or public-coin hash function is a tuple $h, (m, \xi), (m', \xi')$ such that $m \neq m'$, and (h, ξ) and (h, ξ') are valid hash/check pairs for m and m' (respectively). For a chameleon hash function we require the following security property, which intuitively says that it should be hard to find collisions for the hash function even given access to the collision finding algorithm (returning collisions for adaptively chosen hash values). We call such a property *enhanced* collision resistance, and we define it formally below.

Definition 4 (Enhanced collision resistance)**.** Let $\mathcal{CH} = (\mathsf{HGen}, \mathsf{Hash}, \mathsf{HVer}, \mathsf{HCol})$ be a (secret-coin or public-coin) chameleon hash function. We say that $\mathcal{CH}$ satisfies enhanced collision resistance if for all PPT breakers B, the following quantity is negligible in the security parameter:

$$\mathbb{P}\left[\begin{array}{c}(\mathsf{HVer}(hk, m, (h, \xi)) = \mathsf{HVer}(hk, m', (h, \xi')) = 1) \\ \wedge (m \neq m') \wedge (h \notin \mathcal{Q})\end{array} : \begin{array}{c}(h, (m, \xi), (m', \xi')) \leftarrow_{\$} \mathsf{B}^{\mathcal{O}_{hk,tk}(\cdot)}(hk) \\ (hk, tk) \leftarrow_{\$} \mathsf{HGen}(1^{\kappa})\end{array}\right],$$

where the set $\mathcal{Q}$ is the set of all hash values queried by B to its oracle, and oracle $\mathcal{O}_{hk,tk}$ is defined as follows: Upon input a collision query of the form $((h, m, \xi), m')$ run $\mathsf{HVer}(hk, m, (h, \xi)) := d$; if $d = 1$ return the output of $\mathsf{HCol}(tk, (h, m, \xi), m')$, otherwise return $\perp$. In case B is not allowed to query oracle $\mathcal{O}_{hk,tk}$, we simply say that $\mathcal{CH}$ is collision-resistant.

Discussion. Any standard chameleon hash (e.g., the ones considered in [Dam87, KR00, ST01, BR14])is easily seen to imply a public-coin collision-resistant chameleon hash as specified in Definition 3. Let us stress, however, that secret-coin chameleon hash functions can be used for the very same applications as public-coin ones, in particular for constructing chameleon signatures [KR00] and online/offline signatures [EGM96, ST01, BCR+13];the only difference is that one needs to store the check value ξ (instead of the randomness r) in order to verify a hash value, and the hash verification does not in general consist of re-computing the hash.

Unfortunately, as observed by Ateniese and de Medeiros [AdM04], collision resistance is not sufficient for most of the applications of chameleon hash. The reason is that, while the hash function is indeed collision-resistant, any party seeing a collision for the hash function would be able to find other collisions or even recover the secret trapdoor information. This "key exposure" problem makes chameleon hashes not applicable in many contexts. Enhanced collision resistance, as defined above, precisely addresses such issue as it requires that it should be hard to find collisions even after seeing (polynomially many) collisions.

Yet another flavor of chameleon hashing consists of so-called "labeled" hash functions, where the hash algorithm takes as input an additional value λ called the label. Some of these constructions, e.g. the ones in [AdM04, CZK04, CTZD10, CZS+10], do not suffer from the key exposure problem, as they satisfy the property that it should be unfeasible to find collisions

for a "fresh" label λ^*, even given access to an oracle that outputs collisions for arbitrary other labels $\lambda \neq \lambda^*$. [1]However, labeled chameleon hash functions are not useful for constructing online/offline signatures and for the type of application considered in this paper.

4.2 Ingredients

4.2.1 Public-Key Encryption

A Public-Key Encryption (PKE) scheme is a tuple of efficient algorithms $\mathcal{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ defined as follows. (i) The probabilistic algorithm KGen takes as input the security parameter $\kappa \in \mathbb{N}$, and outputs a public/secret key pair (pk, sk). (ii) The probabilistic algorithm Enc takes as input the public key pk, a message $m \in \mathcal{M}$, and implicit randomness $\rho \in \mathcal{R}_{\text{pke}}$, and outputs a ciphertext $c = \mathsf{Enc}(pk, m; \rho)$. the set of all ciphertexts is denoted by $\mathcal{C}$. (iii) The deterministic algorithm Dec takes as input the secret key sk and a ciphertext $c \in \mathcal{C}$ and outputs $m = \mathsf{Dec}(sk, c)$ which is either equal to some message $m \in \mathcal{M}$ or to an error symbol $\perp$.

Correctness. A PKE scheme meets the correctness property if the decryption of a ciphertext encrypting a given plaintext yields the plaintext.

Definition 5 (Correctness for PKE)**.** We say that $\mathcal{PKE}$ satisfies correctness if for all $(pk, sk) \leftarrow\!\!\$\ \mathsf{KGen}(1^\kappa)$ there exists a negligible function $\nu : \mathbb{N} \to [0,1]$ such that that $\mathbb{P}[\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m] \geq 1 - \nu(\kappa)$ (where the randomness is taken over the internal coin tosses of algorithm Enc).

Semantic security. The standard security notion for PKE schemes goes under the name of security against chosen-plaintext attacks (CPA), and informally states that no efficient adversary given the public key can distinguish the encryption of two (possibly known) messages [GM84].

Definition 6 (CPA security)**.** Let $\mathcal{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a PKE scheme. We say that $\mathcal{PKE}$ is CPA-secure if for all PPT adversaries A the following quantity is negligible:

$$\mathbb{P}\left[b' = b : \begin{array}{c} b' \leftarrow\!\!\$\ \mathsf{A}(pk, c); c \leftarrow\!\!\$\ \mathsf{Enc}(pk, m_b); b \leftarrow\!\!\$\ \{0,1\} \\ (m_0, m_1) \leftarrow\!\!\$\ \mathsf{A}(pk); (pk, sk) \leftarrow\!\!\$\ \mathsf{KGen}(1^\kappa) \end{array}\right].$$

4.2.2 Non-Interactive Zero-Knowledge

Let $R : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be an *NP* relation on pairs (x, y), with corresponding language $L := \{y : \exists x \text{ s.t. } R(x, y) = 1\}$. A non-interactive argument for R allows a prover P to convince a verifier V that a common element y belongs to the language L (where both P and V are modeled as PPT algorithms); the prover P is facilitated by knowing a witness x for $y \in L$.

Definition 7 (Non-interactive argument)**.** A non-interactive argument for an *NP* relation R is a tuple of efficient algorithms $\mathcal{NIA} = (\mathsf{I}, \mathsf{P}, \mathsf{V})$ specified as follows.

- $\omega \leftarrow\!\!\$\ \mathsf{I}(1^\kappa)$: The probabilistic algorithm I takes as input the security parameter $\kappa \in \mathbb{N}$, and outputs the public common reference string (CRS) ω.
- $\pi \leftarrow\!\!\$\ \mathsf{P}(\omega, x, y)$: The probabilistic algorithm P takes as input the CRS ω and a pair x, y such that $R(x, y) = 1$, and returns a proof π for membership of $y \in L$.
- $d = \mathsf{V}(\omega, y, \pi)$: The deterministic algorithm V takes as input the CRS ω and a pair (y, π), and returns a decision bit $d \in \{0,1\}$.

[1]Identity-based chameleon hash functions [AdM04, CZT+11, CZS+14] also partially address the key exposure problem, but they require a trusted party and thus only offer a partial solution.

Non-interactive arguments typically satisfy three properties known as completeness, zero-knowledge, and soundness, which we review below. We remark that the CRS is necessary for achieving non-interactive zero-knowledge (see, e.g., [Gol01]).

Completeness. The completeness property states that a honest prover (holding a valid witness x) should always be able to convince the verifier that $y \in L$.

Definition 8 (Completeness for arguments)**.** Let $\mathcal{NIA} = (\mathsf{I}, \mathsf{P}, \mathsf{V})$ be a non-interactive argument for an *NP* relation R. We say that $\mathcal{NIA}$ satisfies completeness if for all pairs (x, y) such that $R(x, y) = 1$, there exists a negligible function $\nu : \mathbb{N} \to [0, 1]$ such that

$$\mathbb{P}\left[\mathsf{V}(\omega, y, \pi) = 1 : \ \pi \leftarrow\!\!\$\ \mathsf{P}(\omega, x, y); \omega \leftarrow\!\!\$\ \mathsf{I}(1^\kappa)\right] \geq 1 - \nu(\kappa).$$

Zero-knowledge. The zero-knowledge property informally says that a possibly malicious verifier cannot acquire any knowledge on the witness that it couldn't acquire by itself. Non-interactive zero-knowledge (NIZK) was first formalized by Blum, Feldman and Micali [BFM88].

Definition 9 (Zero-knowledge)**.** Let $\mathcal{NIA} = (\mathsf{I}, \mathsf{P}, \mathsf{V})$ be a non-interactive argument for an *NP* relation R. We say that $\mathcal{NIA}$ satisfies zero-knowledge if there exists a PPT simulator $\mathsf{S} := (\mathsf{S}_1, \mathsf{S}_2)$ such that for all adversaries A the following quantity is negligible:

$$\left|\mathbb{P}\left[b = b' : \begin{array}{c} b' \leftarrow\!\!\$\ \mathsf{A}(\omega, \tau, \pi_b); \pi_0 \leftarrow\!\!\$\ \mathsf{P}(\omega, x, y); \pi_1 \leftarrow\!\!\$\ \mathsf{S}_2(\tau, y) \\ b \leftarrow\!\!\$\ \{0,1\}; (x, y) \leftarrow\!\!\$\ \mathsf{A}(\omega, \tau); (\omega, \tau) \leftarrow\!\!\$\ \mathsf{S}_1(1^\kappa) \end{array}\right] - \frac{1}{2}\right|.$$

Simulation extractability. The soundness property states that it is hard for a malicious prover to generate an accepting proof π for an element $y \notin L$. Below, we review a strictly stronger formulation of the soundness requirement which is known as simulation extractability, and informally says that soundness still holds even if the malicious prover can access simulated proofs for *true* statements.

This leads to the concept of true-simulation extractable (tSE) NIZK, as defined by Dodis, Haralambiev, Lòpez-Alt, and Wichs [DHLW10].

Definition 10 (True-simulation extractability)**.** Let $\mathcal{NIA} = (\mathsf{I}, \mathsf{P}, \mathsf{V})$ be a NIZK for an *NP* relation R, with zero-knowledge simulator $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$, and let f be an efficiently computable function. We say that $\mathcal{NIA}$ satisfies true-simulation f-extractability (f-tSE for short) if there exists a PPT extractor E such that for all PPT adversaries A the following quantity is negligible:

$$\mathbb{P}\left[\begin{array}{c} y^* \notin \mathcal{Q} \wedge (\mathsf{V}(\omega, y^*, \pi^*) = 1) \\ \wedge \forall x^* \text{ s.t. } f(x^*) = z^*(R(x^*, y^*) = 0) \end{array} : \begin{array}{c} z^* \leftarrow\!\!\$\ \mathsf{E}(\tau, y^*, \pi^*) \\ (y^*, \pi^*) \leftarrow\!\!\$\ \mathsf{A}^{\mathcal{O}_\tau(\cdot,\cdot)}(\omega) \\ (\omega, \tau) \leftarrow\!\!\$\ \mathsf{S}_1(1^\kappa) \end{array}\right],$$

where oracle $\mathcal{O}_\tau$ takes as input pairs (x_i, y_i) and returns the same as $\mathsf{S}_2(\tau, y_i)$ as long as $R(x_i, y_i) = 1$ (and $\perp$ otherwise), and $\mathcal{Q}$ is the set of all values y_i asked to oracle $\mathcal{O}_\tau$.

Note that in the above definition the adversary is only allowed to see simulated proof for *true* statements. A stronger variant (which is not needed in this paper) requires that simulation extractability holds even if the adversary is allowed to see simulated proofs for possibly *false* statements. The latter property is also known under the name of robust NIZK [SCO+01, Gro06].

As noted in [DHLW10] tSE NIZK are significantly more efficient to construct, indeed they can be generically obtained combining any standard NIZK (such as the powerful Groth-Sahai NIZK [GS08]) with a CCA-secure PKE scheme.

4.3 Generic Transformation

To the best of our knowledge, the only construction of a chameleon hash function satisfying *enhanced* collision resistance is due to [AdM04]; the construction is ad-hoc and relies on the Nyberg-Rueppel signature scheme [NR94] (whose security can be shown under the Discrete Logarithm assumption in the generic group model [Sho97]).

Previously to our work it was unknown whether enhanced collision resistance can be achieved in a non ad-hoc fashion, based on different complexity assumptions in the standard model. We answer this open question in the affirmative, by exhibiting a generic transformation from any public-coin collision-resistant chameleon hash to a secret-coin chameleon hash satisfying the stronger *enhanced* collision resistance requirement. The transformation is based on a CPA-secure PKE scheme (cf. Section 4.2.1) and on a tSE NIZK [DHLW10] (cf. Section 4.2.2), and is presented in detail below.

The transformation. Let $\mathcal{CH} = (\mathsf{HGen}, \mathsf{Hash}, \mathsf{HCol})$ be a public-coin chameleon hash function (with message space $\mathcal{M}_{\text{hash}}$ and randomness space $\mathcal{R}_{\text{hash}}$), let $\mathcal{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a PKE scheme (with message space $\mathcal{R}_{\text{hash}}$ and randomness space $\mathcal{R}_{\text{pke}}$), and let $\mathcal{NIA} = (\mathsf{I}, \mathsf{P}, \mathsf{V})$ be a non-interactive argument system for the language

$$L_{\mathcal{CH}} = \{(pk, c, hk, h, m) : \exists (r, \rho) \text{ s.t. } h = \mathsf{Hash}(hk, m; r) \wedge c = \mathsf{Enc}(pk, r; \rho)\}. \tag{1}$$

Consider the secret-coin chameleon hash function $\mathcal{CH}^* = (\mathsf{HGen}^*, \mathsf{Hash}^*, \mathsf{HVer}^*, \mathsf{HCol}^*)$ specified as follows.

- $\mathsf{HGen}^*(1^\kappa)$: Run $(hk, tk) \leftarrow\!\!\$\ \mathsf{HGen}(1^\kappa)$, sample $(pk, sk) \leftarrow\!\!\$\ \mathsf{KGen}(1^\kappa)$, and $\omega \leftarrow\!\!\$\ \mathsf{I}(1^\kappa)$. Return the pair (hk^*, tk^*), such that $hk^* := (hk, \omega, pk)$, and $tk^* := (tk, sk)$.
- $\mathsf{Hash}^*(hk^*, m)$: Sample a random value $r \in \mathcal{R}_{\text{hash}}$ and run $\mathsf{Hash}(hk, m; r) := h$. Sample a random value $\rho \in \mathcal{R}_{\text{pke}}$ and run $c := \mathsf{Enc}(pk, r; \rho)$. Compute the proof $\pi \leftarrow\!\!\$\ \mathsf{P}(\omega, x, y)$, where $x := (r, \rho)$ and $y := (pk, c, hk, h, m)$, and return (h, ξ) such that $\xi := (c, \pi)$.
- $\mathsf{HVer}^*(hk^*, m, (h, \xi))$: Parse $\xi = (c, \pi)$ and return the output of $\mathsf{V}(\omega, y, \pi)$ where $y = (pk, c, hk, h, m)$.
- $\mathsf{HCol}^*(tk^*, (h, m, \xi), m')$: First run $\mathsf{HVer}(hk^*, m, (h, \xi)) := d$; if $d = 0$ then output $\perp$, otherwise, decrypt the randomness $r := \mathsf{Dec}(sk, c)$, compute a collision $r' \leftarrow\!\!\$\ \mathsf{HCol}(tk, (h, m, r), m')$, sample a random $\rho' \in \mathcal{R}_{\text{pke}}$ and encrypt the new randomness $c' := \mathsf{Enc}(pk, r'; \rho')$. Compute the proof $\pi' \leftarrow\!\!\$\ \mathsf{P}(\omega, x', y')$, such that $x' = (r', \rho')$ and $y' := (pk, c', hk, h, m')$, and return $\xi' := (c', \pi')$.

The correctness property follows readily from the correctness of the underlying building blocks. As for security, we show the following result.

Theorem 1. *Assume that $\mathcal{CH}$ is a public-coin collision-resistant chameleon hash function, that $\mathcal{PKE}$ is a CPA-secure PKE scheme, and that $\mathcal{NIA}$ is an f-tSE-NIZK for the language of Eq. (1), where for any witness (r, ρ) we define $f(r, \rho) = r$. Then the above defined secret-coin chameleon hash function $\mathcal{CH}^*$ satisfies enhanced collision resistance.*

Proof. The proof is by game hopping. We define a series of games, starting with the original game for enhanced collision resistance of our construction $\mathcal{CH}^*$. Next, we argue that each pair of adjacent games is computationally indistinguishable and additionally that any PPT breaker has only a negligible advantage in the last game; this yields the theorem.

Below we give a concise description of the games, focusing only on the incremental changes between each game and the previous one; a full description of the games appears in Fig. 5.

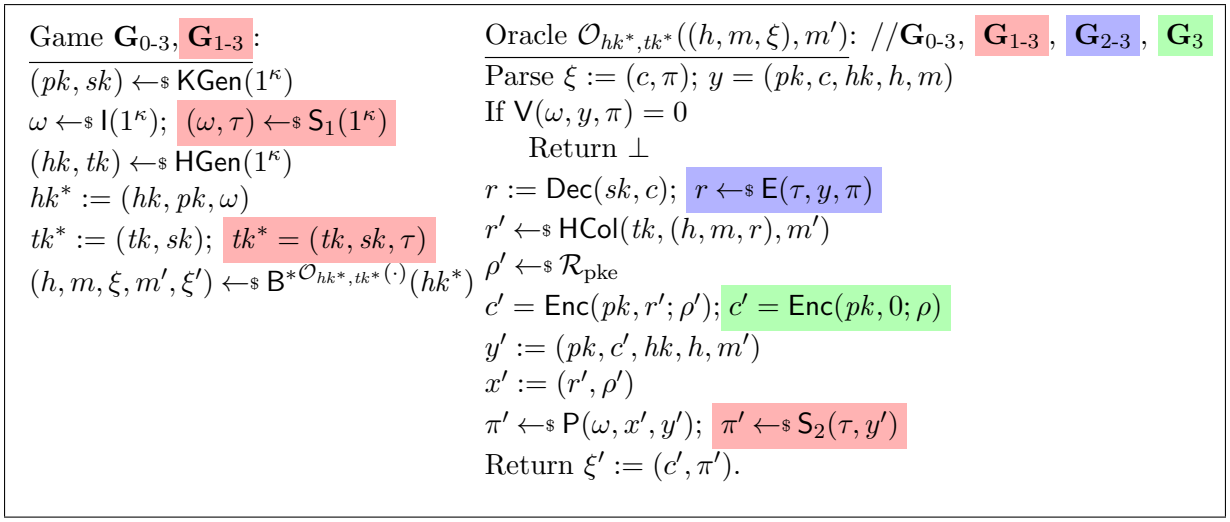


Figure 5: Games in the proof of Theorem 1.

Game $\mathbf{G}_0$: This is the original experiment of Definition 4, running with our secret-coin chameleon hash function $\mathcal{CH}^*$ and a PPT breaker B^*.

Game $\mathbf{G}_1$: We change the way collision queries are answered. In particular, we compute the proof π' by running the zero-knowledge simulator $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$ instead of running the real prover P. In order to do so, we first set-up the CRS by running $(\omega, \tau) \leftarrow\$ \mathsf{S}_1(1^\kappa)$ and later generate π' by running $\mathsf{S}_2(\tau, \cdot)$.

Game $\mathbf{G}_2$: We change the way collision queries are answered. In particular, instead of recovering the randomness r by decrypting the ciphertext c, we now compute r by extracting the proof π. In order to do so, we first set-up the CRS by running $(\omega, \tau) \leftarrow\$ \mathsf{S}_1(1^\kappa)$ and later recover r by running $\mathsf{E}(\tau, \cdot, \cdot)$.

Game $\mathbf{G}_3$: We change the way collision queries are answered. In particular, instead of first equivocating the hash value h yielding some new randomness r' (corresponding to message m') and then computing c' as an encryption of r', we simply let c' be an encryption of zero.

In each game $\mathbf{G}_i$, for $i \in [0, 3]$, we define the event $\mathbf{G}_i = 1$ to be the event that B^* wins in the corresponding game, namely that the tuple (h, m, ξ, m', ξ') returned by B^* is such that $m \neq m'$ and both proofs π and π' contained in ξ and ξ' are accepting. Next, we analyze the computational distance between each pair of adjacent games.

Claim 1. *For all PPT distinguishers* D *there exists a negligible function* $\nu_{0,1} : \mathbb{N} \to [0, 1]$ *such that* $|\mathbb{P}[\mathsf{D}(\mathbf{G}_0(\kappa)) = 1] - \mathbb{P}[\mathsf{D}(\mathbf{G}_1(\kappa)) = 1]| \leq \nu_{0,1}(\kappa)$.

Proof of claim. Assume that there exists a PPT distinguisher D and a polynomial $p_{0,1}(\cdot)$ such that, for infinitely many values of $\kappa \in \mathbb{N}$, we have that D distinguishes between game $\mathbf{G}_0$ and game $\mathbf{G}_1$ with probability at least $1/p_{0,1}(\kappa)$. Let $q \in poly(\kappa)$ be the number of queries that D is allowed to ask to its oracle. For an index $i \in [0, q]$ consider the hybrid game $\mathbf{H}_i$ that answers the first i queries as in game $\mathbf{G}_0$ and all the subsequent queries as in game $\mathbf{G}_1$. Note that $\mathbf{H}_0 \equiv \mathbf{G}_1$ and $\mathbf{H}_q \equiv \mathbf{G}_0$.

By a standard hybrid argument, we have that there exists an index $i \in [0, q]$ such that D tells apart $\mathbf{H}_{i-1}$ and $\mathbf{H}_i$ with non-negligible probability $1/q \cdot 1/p_{0,1}(\kappa)$. We build a PPT adversary A that (using distinguisher D) breaks the non-interactive zero-knowledge property of $\mathcal{NIA}$. A formal description of A follows.

Adversary A:

- The challenger runs $(\omega, \tau) \leftarrow\!\!\$\ \mathsf{S}_1(1^\kappa)$ and forwards (ω, τ) to A.
- Run $(hk, tk) \leftarrow\!\!\$\ \mathsf{HGen}(1^\kappa)$, sample $(pk, sk) \leftarrow\!\!\$\ \mathsf{KGen}(1^\kappa)$, and send $hk^* := (hk, pk, \omega)$ to D.
- Upon input a collision query of type $((h_j, m_j, \xi_j), m'_j)$ from D, such that $\xi_j = (c_j, \pi_j)$, first check whether $\mathsf{V}(\omega, (pk, c_j, hk, h_j, m), \pi_j) = 0$. In case this happens return $\perp$, otherwise: Decrypt the randomness $r_j := \mathsf{Dec}(sk, c_j)$, find a collision $r'_j \leftarrow\!\!\$\ \mathsf{HCol}(tk, (h_j, m_j, r_j), m'_j)$, sample $\rho'_j \leftarrow\!\!\$\ \mathcal{R}_{\mathrm{pke}}$, and let $c'_j = \mathsf{Enc}(pk, r'_j; \rho'_j)$. Hence:
  - If $j \leq i-1$, compute $\pi' \leftarrow\!\!\$\ \mathsf{P}(\omega, (pk, c'_j, hk, h_j m'_j), (r'_j, \rho'_j))$ and return $\xi'_j := (c'_j, \pi'_j)$ to D.
  - If $j = i$, forward $((r'_j, \rho'_j), (pk, c'_j, hk, h_j, m'_j))$ to the challenger obtaining a proof π'_j; return $\xi'_j := (c'_j, \pi'_j)$ to D.
  - If $j \geq i+1$, compute $\pi'_j \leftarrow\!\!\$\ \mathsf{S}_2(\tau, (pk, c'_j, hk, h_j, m'_j))$ and return $\xi' := (c'_j, \pi'_j)$.
- Output whatever D outputs.

For the analysis, note that the only difference between game $\mathbf{H}_{i-1}$ and game $\mathbf{H}_i$ is on how the i-th query is answered. In particular, in case the hidden bit b in the definition of non-interactive zero-knowledge equals zero A's simulation produces exactly the same distribution as in $\mathbf{H}_{i-1}$, and otherwise A's simulation produces exactly the same distribution as in $\mathbf{H}_i$. Hence, A breaks the NIZK property with non-negligible advantage $1/q \cdot 1/p_{0,1}(\kappa)$, a contradiction. This concludes the proof. □

Claim 2. *For all PPT distinguishers* D *there exists a negligible function* $\nu_{1,2} : \mathbb{N} \to [0,1]$ *such that* $|\mathbb{P}[\mathsf{D}(\mathbf{G}_1(\kappa)) = 1] - \mathbb{P}[\mathsf{D}(\mathbf{G}_2(\kappa)) = 1]| \leq \nu_{1,2}(\kappa)$.

Proof of claim. Let $q \in poly(\kappa)$ be the number of collision queries that the adversary is allowed to ask to its oracle, where each query has a type $((h_j, m_j, \xi_j), m'_j)$ for some $\xi'_j = (c_j, \pi_j)$. Define the following "bad event" E, in the probability space of game $\mathbf{G}_1$: The event becomes true if there exists an index $i \in [q]$ such that the proof π_i is accepting for $(pk, c_i, hk, h_i, m_i) \in L_{\mathcal{CH}^*}$, but running the extractor $\mathsf{E}(\tau, \cdot, \cdot)$ on (y_i, π_i) yields a value r_i such that $h_i \neq \mathsf{Hash}(hk, m_i; r_i)$, whereas this does not happen if r_i is computed as in $\mathbf{G}_1$.

Notice that $\mathbf{G}_1(\kappa)$ and $\mathbf{G}_2(\kappa)$ are identically distributed conditioning on E not happening. Hence, by a standard argument, it suffices to bound the probability of provoking event E by all PPT adversaries D. Assume that there exists a PPT distinguisher D and a polynomial $p_{1,2}(\cdot)$ such that, for infinitely many values of $\kappa \in \mathbb{N}$, we have that D provokes event E with probability at least $1/p_{1,2}(\kappa)$.

We build an adversary A that (using distinguisher D) breaks true-simulation extractability of $\mathcal{NIA}$ (for the function f defined in the theorem statement). A formal description of A follows.

Adversary A:

- The challenger runs $(\omega, \tau) \leftarrow\!\!\$\ \mathsf{S}_1(1^\kappa)$ and forwards ω to A.
- Run $(hk, tk) \leftarrow\!\!\$\ \mathsf{HGen}(1^\kappa)$, sample $(pk, sk) \leftarrow\!\!\$\ \mathsf{KGen}(1^\kappa)$, and send $hk^* := (hk, pk, \omega)$ to D.
- Upon input a collision query of type $((h_j, m_j, \xi_j), m'_j)$ from D, such that $\xi_j = (c_j, \pi_j)$, first check whether $\mathsf{V}(\omega, (pk, c_j, hk, h_j, m_j), \pi_j) = 0$. In case this happens return $\perp$, otherwise: Decrypt the randomness $r_j := \mathsf{Dec}(sk, c_j)$, find a

collision $r'_j \leftarrow\!\!\$\ \mathsf{HCol}(tk, (h_j, m_j, r_j), m'_j)$, sample $\rho'_j \leftarrow\!\!\$\ \mathcal{R}_{\text{pke}}$ and encrypt $c'_j := \mathsf{Enc}(pk, r'_j; \rho'_j)$. Forward (x'_j, y'_j) to the target oracle, where $x'_j := (r'_j, \rho'_j)$ and $y'_j := (pk, c'_j, hk, h_j, m'_j)$, obtaining a simulated proof π'_j and forward $\xi'_j := (c'_j, \pi'_j)$.

- After D is done with its queries, sample a random index $i \leftarrow\!\!\$\ [q]$ and forward (y^*, π^*) to the challenger, where the values $y^* := (pk, c_i, hk, h_i, m_i)$ and $\pi^* := \pi_i$ are taken from D's i-th query to the collision oracle.

For the analysis, we note that A's simulation is perfect as the answer to D's queries to the collision oracle are distributed exactly as in $\mathbf{G}_1$. Thus, D provokes event E with probability $1/p_{1,2}(\kappa)$ and so the pair (y^*, π^*) violates the f-tSE property of the non-interactive argument with non-negligible probability $1/q \cdot 1/p_{1,2}(\kappa)$. The claim follows. □

Claim 3. *For all PPT distinguishers* D *there exists a negligible function* $\nu_{2,3} : \mathbb{N} \to [0,1]$ *such that* $|\mathbb{P}[\mathsf{D}(\mathbf{G}_2(\kappa)) = 1] - \mathbb{P}[\mathsf{D}(\mathbf{G}_3(\kappa)) = 1]| \leq \nu_{2,3}(\kappa)$.

Proof of claim. Assume that there exists a PPT distinguisher D and a polynomial $p_{2,3}(\cdot)$ such that, for infinitely many values of $\kappa \in \mathbb{N}$, we have that D distinguishes between game $\mathbf{G}_2$ and game $\mathbf{G}_3$ with probability at least $1/p_{2,3}(\kappa)$. Let $q \in poly(\kappa)$ be the number of queries that D is allowed to ask to its oracle. For an index $i \in [0, q]$ consider the hybrid game $\mathbf{H}_i$ that answers the first i queries as in game $\mathbf{G}_2$ and all the subsequent queries as in game $\mathbf{G}_3$. Note that $\mathbf{H}_0 \equiv \mathbf{G}_3$ and $\mathbf{H}_q \equiv \mathbf{G}_2$.

By a standard hybrid argument, we have that there exists an index $i \in [0, q]$ such that D tells apart $\mathbf{H}_{i-1}$ and $\mathbf{H}_i$ with non-negligible probability $1/q \cdot 1/p_{2,3}(\kappa)$. We build a PPT adversary A that (using distinguisher D) breaks CPA security of $\mathcal{PKE}$. A formal description of A follows.

Adversary A:

- Receive pk from the challenger, where $(pk, sk) \leftarrow\!\!\$\ \mathsf{KGen}(1^\kappa)$.
- Run $(hk, tk) \leftarrow\!\!\$\ \mathsf{HGen}(1^\kappa)$, $(\omega, \tau) \leftarrow\!\!\$\ \mathsf{S}_1(1^\kappa)$, and send $hk^* := (hk, pk, \omega)$ to D.
- Upon input a collision query of type $((h_j, m_j, \xi_j), m'_j)$ from D, such that $\xi_j = (c_j, \pi_j)$, first check whether $\mathsf{V}(\omega, (pk, c_j, hk, h_j, m), \pi_j) = 0$. In case this happens return $\perp$, otherwise: Extract the randomness $r_j := \mathsf{E}(\tau, (pk, c_j, hk, h_j, m), \pi_j)$ and find a collision $r'_j \leftarrow\!\!\$\ \mathsf{HCol}(tk, (h_j, m_j, r_j), m'_j)$. Hence:
  - If $j \leq i-1$, sample a random $\rho'_j \in \mathcal{R}_{\text{pke}}$, encrypt $c'_j := \mathsf{Enc}(pk, r'_j; \rho'_j)$, simulate a proof $\pi'_j \leftarrow\!\!\$\ \mathsf{S}_2(\tau, (pk, c'_j, hk, h_j, m'))$, and return $\xi'_j := (c'_j, \pi'_j)$ to D.
  - If $j = i$, forward $(r'_j, 0)$ for the challenger receiving back a ciphertext c'_j; simulate a proof $\pi'_j \leftarrow\!\!\$\ \mathsf{S}_2(\tau, (pk, c'_j, hk, h_j, m'))$, and return $\xi'_j := (c'_j, \pi'_j)$ to D.
  - If $j \geq i+1$, sample a random $\rho'_j \in \mathcal{R}_{\text{pke}}$, encrypt $c'_j := \mathsf{Enc}(pk, 0; \rho'_j)$, simulate a proof $\pi'_j \leftarrow\!\!\$\ \mathsf{S}_2(\tau, (pk, c'_j, hk, h_j, m'))$, and return $\xi'_j := (c'_j, \pi'_j)$ to D.
- Output whatever D outputs.

For the analysis, note that the only difference between game $\mathbf{H}_{i-1}$ and game $\mathbf{H}_i$ is on how the i-th collision query is answered. In particular, in case the hidden bit b in the definition of CPA security equals zero, A's simulation produces exactly the same distribution as in $\mathbf{H}_{i-1}$, and otherwise A's simulation produces exactly the same distribution as in $\mathbf{H}_i$. Hence, A breaks CPA security with non-negligible advantage $1/q \cdot 1/p_{2,3}(\kappa)$, a contradiction. This concludes the proof. □

Finally, we show that any PPT adversary has a negligible success probability in game $\mathbf{G}_3$, which concludes the proof of the theorem.

Claim 4. *For all PPT breakers* B^* *there exists a negligible function* $\nu_3 : \mathbb{N} \to [0,1]$ *such that* $\mathbb{P}[\mathbf{G}_3(\kappa) = 1] \leq \nu_3(\kappa)$.

Proof of claim. The proof is down to collision resistance of the underlying public-coin chameleon hash function $\mathcal{CH}$. Namely, assume that there is a PPT breaker B^* and a polynomial $p_3(\cdot)$ such that, for infinitely many values of $\kappa \in \mathbb{N}$, we have $\mathbb{P}[\mathbf{G}_3(\kappa) = 1] \geq 1/p_3(\kappa)$. We build a PPT breaker B that (using breaker B^*) breaks collision resistance of $\mathcal{CH}$ as follows.

Adversary B:

- Receive hk from the challenger, where $(hk, tk) \leftarrow\!\!\$\ \mathsf{HGen}(1^\kappa)$.
- Run $(\omega, \tau) \leftarrow\!\!\$\ \mathsf{S}_1(1^\kappa)$, $(pk, sk) \leftarrow\!\!\$\ \mathsf{KGen}(1^\kappa)$, and send $hk^* := (hk, pk, \omega)$ to B^*.
- Upon input a collision query of type $((h_j, m_j, \xi_j), m'_j)$ from B^*, such that $\xi_j = (c_j, \pi_j)$, answer as this would be done in game $\mathbf{G}_3$. Note that this can be done because the way collision queries are treated in $\mathbf{G}_3$ is completely independent on the trapdoor information tk.
  In particular, first check whether $\mathsf{V}(\omega, (pk, c_j, hk, h_j, m), \pi_j) = 0$. In case this happens return $\perp$, otherwise: Extract the randomness $r_j := \mathsf{E}(\tau, (pk, c_j, hk, h_j, m), \pi_j)$, sample a random $\rho'_j \in \mathcal{R}_{\text{pke}}$, encrypt $c'_j := \mathsf{Enc}(pk, 0; \rho'_j)$, and simulate $\pi'_j \leftarrow\!\!\$\ \mathsf{S}_2(\tau, (pk, c'_j, hk, h_j, m'_j))$; return $\xi'_j := (c'_j, \pi'_j)$ to B^*.
- Eventually B^* outputs the tuple (h, m, ξ, m', ξ'). When this happens, let $r := \mathsf{Dec}(sk, r)$ and $r' := \mathsf{Dec}(sk, c')$, and output (h, m, r, m', r') to the challenger.

For the analysis, note that B perfectly simulates B^*'s queries to the collision oracle. It follows that, with probability at least $1/p_3(\kappa)$, adversary B^* outputs a collision for $\mathcal{CH}^*$. This implies that the output of B constitutes a valid collision for $\mathcal{CH}$, with the same probability, and thus concludes the proof of the claim. □

□

4.4 Concrete Instantiations

We now explain how to instantiate our generic transformation from the previous section using standard complexity assumptions. We need three main ingredients: (i) A public-coin chameleon hash function $\mathcal{CH} = (\mathsf{HGen}, \mathsf{Hash}, \mathsf{HCol})$ with randomness space $\mathcal{R}_{\text{hash}}$; (ii) A CPA-secure PKE scheme $\mathcal{PKE}^1 = (\mathsf{KGen}^1, \mathsf{Enc}^1, \mathsf{Dec}^1)$ with message space $\mathcal{M}^1_{\text{pke}} := \mathcal{R}_{\text{hash}}$ and randomness space $\mathcal{R}^1_{\text{pke}}$; (iii) An f-tSE NIZK for the language of Eq. (1), where the function $f : \mathcal{R}_{\text{hash}} \times \mathcal{R}^1_{\text{pke}} \to \mathcal{R}_{\text{hash}}$ has a type $f(r, \rho) = r$. For the latter component, we rely on the construction due to Dodis *et al.* [DHLW10] that allows to obtain an f-tSE NIZK for any efficiently computable function f and for any language L, based on a standard (non-extractable) NIZK for that language and a CCA-secure PKE scheme.

Let $\mathcal{PKE}^2 = (\mathsf{KGen}^2, \mathsf{Enc}^2, \mathsf{Dec}^2)$ be a CCA-secure PKE scheme with message space $\mathcal{M}^2_{\text{pke}} := \mathcal{R}_{\text{hash}}$. Plugging in the construction from [DHLW10] the check value ξ in our construction has the form $\xi := (c_1, c_2, \pi)$, where π is a standard NIZK argument for $((pk_1, c_1), (hk, h, m), (pk_2, c_2)) \in L_{\mathcal{CH}}$, with language $L_{\mathcal{CH}}$ being defined as follows:

$$L_{\mathcal{CH}} = \left\{ ((pk_1, c_1), (hk, h, m), (pk_2, c_2)) : \exists (r, \rho_1, \rho_2) \text{ s.t. } \begin{array}{r} h = \mathsf{Hash}(hk, m; r) \\ c_1 = \mathsf{Enc}^1(pk_1, r; \rho_1) \\ c_2 = \mathsf{Enc}^2(pk_2, r; \rho_2) \end{array} \right\}, \quad (2)$$

and where pk_1 and pk_2 are public keys generated via KGen^1 and KGen^2 (respectively), and where hk is generated via HGen.

As for the public-coin chameleon hash function, we use the framework of Bellare and Ristov [BR14] which is based on so-called Sigma-protocols. Below, we first define the complexity assumptions on which we build, and later detail two concrete instantiations (the first one in the random oracle model and the second one in the standard model).

4.4.1 Hardness Assumptions

We review the main complexity assumptions on which our instantiations are based. In what follows, let $\mathbb{G}$ be a group with prime order q and with generator $\mathbb{G}$.

Discrete Logarithm assumption. Let $g \leftarrow_\$ \mathbb{G}$ and $x \leftarrow_\$ \mathbb{Z}_q$. We say that the Discrete Logarithm (DL) assumption holds in $\mathbb{G}$ if it is computationally hard to find $x \in \mathbb{Z}_q$ given $y = g^x \in \mathbb{G}$.

Decisional Diffie-Hellman assumption. Let $g_1, g_2 \leftarrow_\$ \mathbb{G}$ and $x_1, x_2, x \leftarrow_\$ \mathbb{Z}_q$. We say that the Decisional Diffie-Hellman (DDH) assumption holds in $\mathbb{G}$ if the following distributions are computationally indistinguishable: $(\mathbb{G}, g_1, g_2, g_1^{x_1}, g_2^{x_2})$ and $(\mathbb{G}, g_1, g_2, g_1^{x}, g_2^{x})$.

Symmetric External Diffie-Hellman assumption. Let $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ be groups of prime order q and let $\mathsf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a non-degenerate, efficiently computable, bilinear map. The Symmetric External Diffie-Hellman (SXDH) assumption states that the DDH assumption holds in both $\mathbb{G}_1$ and $\mathbb{G}_2$. Such an assumption is not satisfied in case $\mathbb{G}_1 = \mathbb{G}_2$, but it is believed to hold in case there is no efficiently computable mapping between $\mathbb{G}_1$ and $\mathbb{G}_2$ [Sco02, BBS04].

K-Linear assumption [Sha07, HK07]. Let $K \geq 1$ be a constant, and let $g_1, \ldots, g_{K+1} \leftarrow_\$ \mathbb{G}$ and $x_1, \ldots, x_K \leftarrow_\$ \mathbb{Z}_q$. The K-linear assumption holds in $\mathbb{G}$ if the following distributions are computationally indistinguishable: $(\mathbb{G}, g_1^{x_1}, \ldots, g_K^{x_K}, g_{K+1}^{x_{K+1}})$ and $(\mathbb{G}, g_1^{x_1}, \ldots, g_K^{x_K}, g_{K+1}^{\sum_{i=1}^{K} x_i})$. Note that for $K = 1$ we obtain the DDH assumption, and for $K = 2$ we obtain the so-called Linear assumption [Sha07].

In what follows we assume that the K-linear assumption holds in both $\mathbb{G}_1$ and $\mathbb{G}_2$, which is the case for symmetric pairings provided that $K \geq 2$. For $K = 1$ we slightly abuse notation, and assume that the SXDH assumption holds instead (although that requires asymmetric pairings).

4.4.2 Random Oracle Model Instantiation

For concreteness, we focus here on a specific construction relying on the DDH assumption and on the Sigma-protocol due to Schnorr [Sch91]; similar constructions can be obtained based on the RSA assumption, on Quadratic Residuosity, and on Factoring, using the Sigma protocols due to Guillou-Quisquater [GQ88], Fiat-Shamir [FS86], Ong-Schnorr [OS90], Okamoto [Oka92], and Fischlin and Fischlin [FF02].

Public-Coin Hash: Let $\mathbb{G}$ be a group with prime order q and with generator g, where the Discrete Logarithm problem is believed to be hard. Algorithm HGen picks a random $x \leftarrow_\$ \mathbb{Z}_q$ and defines $hk := g^x = y$ and $tk = x$. In order to hash a message $m \in \mathbb{Z}_q$, algorithm $\mathsf{Hash}(hk, m; r)$ picks a random $r \leftarrow_\$ \mathbb{Z}_q$ and returns $h := g^r \cdot y^{-m}$. In order to

compute a collision, algorithm $\mathsf{HCol}(tk,(h,m,r),m')$ returns $r'=r-x\cdot(m-m')$ mod q.[2] Notice that $\mathcal{R}_{\text{hash}}=\mathcal{M}_{\text{hash}}:=\mathbb{Z}_q$.

CPA PKE: We use the ElGamal PKE scheme [ElG85]. In order to encrypt messages in $\mathbb{Z}_q$ we rely on a public, onto, invertible mapping $\Omega:\mathbb{Z}_q\to\mathbb{G}$.[3] Let $\hat{g}$ be a generator in $\mathbb{G}$. The public-key is $pk_1=\hat{g}^{\hat{x}}=\hat{y}$ for a random secret key $\hat{x}\leftarrow_{\$}\mathbb{Z}_q$, and the encryption of $r\in\mathbb{Z}_q$ is equal to $c_1:=(c_1^1,c_1^2)=(\hat{g}^{\rho_1},\hat{y}^{\rho_1}\cdot\Omega(r))$ for a random $\rho_1\leftarrow_{\$}\mathbb{Z}_q$.

CCA PKE: We use the PKE scheme due to Cramer and Shoup [CS98]. In order to encrypt messages in $\mathbb{Z}_q$ we rely on the same mapping Ω described above. Let g_1,g_2 be generators in $\mathbb{G}$, and let $H:\mathbb{G}^3\to\mathbb{Z}_q$ be a standard collision-resistant hash function. The public-key is $pk_2=(g_1^{x_1^1}g_2^{x_2^1},g_1^{x_1^2}g_2^{x_2^2},g_1^{x_1^3}g_2^{x_2^3})=(y_1,y_2,y_3)$ for a random secret key $(x_1^1,x_2^1,x_1^2,x_2^2,x_1^3,x_2^3)\leftarrow_{\$}\mathbb{Z}_q^6$, and the encryption of $r\in\mathbb{Z}_q$ is equal to $c_2:=(c_2^1,c_2^2,c_2^3,c_2^4)=(g_1^{\rho_2},g_2^{\rho_2},y_3^{\rho_2}\cdot\Omega(r),y_1^{\rho_2}\cdot y_2^{\rho_2\cdot t})$ for a random $\rho_2\leftarrow_{\$}\mathbb{Z}_q$ and with $t=H(c_2^1,c_2^2,c_2^3)$.

NIZK: We use the Fiat-Shamir heuristic [FS86].[4] Let $G:\{0,1\}^*\to\mathbb{Z}_q$ be a hash function modeled as a random oracle. The language of Eq. (2) boils down to prove knowledge of $(r,\rho_1,\rho_2)\in\mathbb{Z}_q^3$ such that: (i) $r=\log_g(h\cdot y^m)$; (ii) $\rho_1=\log_{\hat{g}}c_1^1$; (iii) $\log_{g_1}c_2^1=\rho_2=\log_{g_2}c_2^2$; (iv) $c_2^4=y_1^{\rho_2}\cdot y_2^{\rho_2\cdot t}$; (v) $c_1^2/c_2^3=\hat{y}^{\rho_1}\cdot y_3^{-\rho_2}$. Notice that this is indeed sufficient, as proving knowledge of ρ_1, ρ_2 implies knowledge of $\Omega(r)=c_1^2/\hat{y}_1^{\rho}=c_2^3/y_3^{\rho_2}$ (and, in turn, $\Omega(r)$ uniquely determines r).

The proofs in (i) and (ii) can be obtained from a Sigma-protocol for showing knowledge of a discrete logarithm [Sch91]. The proof in (iii) can be obtained from a Sigma-protocol for showing equality of two discrete logarithms [CP92]. The proof in (iv) and (v) can be obtained from a Sigma-protocol for showing knowledge of a representation [Oka92].

Hence, the NIZK π has a type $\pi:=(\pi_0,\pi_1,\pi_2,\pi_3,\pi_4)$ where: (i) $\pi_0=(\alpha,\gamma)=(g^a,\beta r+a)$ for random $a\leftarrow_{\$}\mathbb{Z}_q$ and $\beta=:G(h\cdot y^m||\alpha)$; (ii) $\pi_1=(\alpha_1,\gamma_1)=(\hat{g}^{a_1},\beta_1\rho_1+a_1)$ for random $a_1\leftarrow_{\$}\mathbb{Z}_q$ and $\beta_1=:G(c_1^1||\alpha_1)$; (iii) $\pi_2=(\alpha_2^1,\alpha_2^2,\gamma_2)=(g_1^{a_2},g_2^{a_2},\beta_2\cdot\rho_2+a_2)$ for random $a_2\leftarrow_{\$}\mathbb{Z}_q$ and $\beta_2=:G(c_2^1||c_2^2||\alpha_2^1||\alpha_2^2)$; (iv) $\pi_3=(\alpha_3,\gamma_3^1,\gamma_3^2)=(y_1^{a_3^1}\cdot y_2^{a_3^2},\beta_3\cdot\rho_2+a_3^1,\beta_3\cdot\rho_2\cdot t+a_3^2)$, for random $a_3^1,a_3^2\leftarrow_{\$}\mathbb{Z}_q$ and $\beta_3=(c_2^4||y_1||y_2||\alpha_3)$; (v) $\pi_4=(\alpha_4,\gamma_4^1,\gamma_4^2)=(\hat{y}^{a_4^1}\cdot y_3^{a_4^2},\beta_4\cdot\rho_1+a_4^1,-\beta_4\cdot\rho_2+a_4^2)$, for random $a_4^1,a_4^2\leftarrow_{\$}\mathbb{Z}_q$ and $\beta_4=(c_1^2/c_2^3||\hat{y}||y_3||\alpha_4)$.[5]

Putting together the above constructions we obtain the following result.

Corollary 1. *Let $\mathbb{G}$ be a group with prime order q. Under the DDH assumption in $\mathbb{G}$ there exists a secret-coin chameleon hash function satisfying enhanced collision resistance in the ROM, such that the hash value consists of a single element of $\mathbb{G}$, whereas the check value consists of 12 elements of $\mathbb{G}$ plus 7 elements of $\mathbb{Z}_q$.*

4.4.3 Standard Model Instantiation

We give an instantiation based on the K-Linear assumption.

Let $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ be groups of prime order q and let $\mathsf{e}:\mathbb{G}_1\times\mathbb{G}_2\to\mathbb{G}_T$ be a non-degenerate, efficiently computable, bilinear map. Before describing the ingredients, we briefly recall the

[2] For readers familiar with [BR14], the hashing algorithm corresponds to the (strong) HVZK simulator and the collision-finding algorithm corresponds to the strong prover of the underlying Sigma-protocol.

[3] As observed in [CPP06] relying on such an encoding might be computationally expensive; we can adopt the same technique as in [CPP06] for obtaining an encoding-free solution.

[4] It might seem that using the construction of Dodis *et al.* [DHLW10] for obtaining a tSE NIZK in the ROM is an overkill, as the Fiat-Shamir heuristic directly yields a full-fledged simulation-sound extractable NIZK. However, the Fiat-Shamir transform is only known to satisfy a weaker form of extractability [FKMV12] which is insufficient for our application. Alternatively, we could use Fischlin's transformation [Fis05], but this would probably result in longer proofs.

[5] For simplicity, we omit the description of the verification algorithm.

Groth-Sahai [GS08] proof system for showing multi-exponentiation equations. The CRS consists of vectors $\vec{u}_1, \dots \vec{u}_K, \vec{u}$ such that $\vec{u}_i = (u_0, 1, \dots, 1, u_i, 1, \dots, 1) \in \mathbb{G}_1^{K+1}$, where $u_0, u_1, \dots, u_K$ are randomly chosen in $\mathbb{G}_1^*$ and $\vec{u}$ is randomly chosen in the span of $(\vec{u}_1, \dots, \vec{u}_K)$.

Consider an equation of type $\tilde{g}_0 = \tilde{g}_1^{\varphi_1} \cdot, \cdots, \cdot \tilde{g}_N^{\varphi_N}$, where $\tilde{g}_0, \tilde{g}_1, \dots, \tilde{g}_N \in \mathbb{G}_2$ are constants and $\varphi_1, \dots, \varphi_N \in \mathbb{Z}_q$ are variables. To generate a proof one first commits to all the variables one by one; in particular, to commit to $\varphi_i \in \mathbb{Z}_q$, we sample $\vec{s}_i = (s_i^1, \dots, s_i^K) \leftarrow\!\!\$\ \mathbb{Z}_q^K$ and compute $\vec{\psi}_i := \vec{u}^{\varphi} \prod_{j=1}^K \vec{u}_j^{s_i^j} \in \mathbb{G}_1^{K+1}$ (where vector multiplication is defined component-wise). Hence, we return the proof elements $\pi_j = \prod_{i=1}^N \tilde{g}_i^{s_i^j} \in \mathbb{G}_2$ for $j \in [K]$. In order to verify a proof $\pi = (\vec{\psi}_1, \dots, \vec{\psi}_N, \pi_1, \dots, \pi_K)$, we check that

$$\prod_{i=1}^N \hat{\mathsf{e}}(\vec{\psi}_i, \tilde{g}_i) = \hat{\mathsf{e}}(\vec{u}, \tilde{g}_0) \prod_{j=1}^K \vec{u}_j \cdot \pi_j,$$

where $\hat{\mathsf{e}} : \mathbb{G}_1^{K+1} \times \mathbb{G}_2 \to \mathbb{G}_T^{K+1}$, such that $\hat{\mathsf{e}}((a_1, \dots, a_{K+1}), b) := (\mathsf{e}(a_1, b), \dots, \mathsf{e}(a_{K+1}, b))$, is a bilinear map.

Public-Coin Hash: We use the same public-coin chameleon hash function described in the previous section, based on the DL assumption in $\mathbb{G}_2$ (which is implied by both the SXDH assumption and the DLIN assumption). Recall that $\mathcal{R}_{\text{hash}} = \mathcal{M}_{\text{hash}} := \mathbb{Z}_q$, whereas $\mathbb{G}_2$ is the output range of the hash function.

CPA PKE: We use the Linear ElGamal PKE scheme, introduced by Boneh, Boyen and Shacham [BBS04], which is based on the K-Linear assumption. Let Ω be as above. At key generation we sample a random generator $\hat{g}_{K+1} \in \mathbb{G}_2$, and, for all $i \in [K]$, define $\hat{g}_i := \hat{g}_{K+1}^{1/\hat{x}_i}$ for random $\hat{x}_i \in \mathbb{Z}_q$; the public key is $pk_1 = (\hat{g}_1, \dots, \hat{g}_{K+1})$, and the secret key is $(\hat{x}_1, \dots, \hat{x}_K)$. The encryption of $r \in \mathbb{Z}_q$ is equal to $c_1 = (c_1^1, \dots, c_1^K, c_1^{K+1})$ such that $c_1^i := \hat{g}_i^{\rho_1^i}$ for random $\rho_1^i \leftarrow\!\!\$\ \mathbb{Z}_q$ and for all $i \in [K]$, and $c_1^{K+1} := \Omega(r) \cdot \hat{g}_{K+1}^{\sum_{i=1}^K \rho_1^i}$.

CCA PKE: We use the Linear Cramer-Shoup PKE scheme, introduced by Shacham [Sha07], which is based on the K-Linear assumption. Let $H : \mathbb{G}_2^{K+2} \to \mathbb{Z}_q$ be a collision-resistant hash function. At key generation we sample random generators $g_1, \dots, g_{K+1} \in \mathbb{G}_2$, and random exponents $x_i^j \in \mathbb{Z}_q$ for all $i \in [K+1]$ and for all $j \in [3]$. We then compute:

$$\begin{array}{lll}
y_1^1 := g_1^{x_1^1} g_2^{x_2^1} & y_2^1 := g_1^{x_1^2} g_2^{x_2^2} & y_3^1 := g_1^{x_1^3} g_2^{x_2^3} \\
y_1^2 := g_1^{x_1^1} g_3^{x_3^1} & y_2^2 := g_1^{x_1^2} g_3^{x_3^2} & y_3^2 := g_1^{x_1^3} g_3^{x_3^3} \\
\vdots & \vdots & \vdots \\
y_1^K := g_1^{x_1^1} g_{K+1}^{x_{K+1}^1} & y_2^K := g_1^{x_1^2} g_{K+1}^{x_{K+1}^2} & y_3^K := g_1^{x_1^3} g_{K+1}^{x_{K+1}^3},
\end{array}$$

and return public key $pk_2 := (g_1, \dots, g_{K+1}, y_1^1, \dots, y_1^K, y_1^2, \dots, y_2^K, y_1^3, \dots, y_3^K)$ with corresponding secret key $(x_1^1, \dots, x_{K+1}^1, x_1^2, \dots, x_{K+1}^2, x_1^3, \dots, x_{K+1}^3) \in \mathbb{Z}_q^{3K+3}$. In order to encrypt $r \in \mathbb{Z}_q$ one samples $\rho_2^1, \dots, \rho_2^K \leftarrow\!\!\$\ \mathbb{Z}_q$ and returns a ciphertext:

$$\begin{aligned}
c_2 &:= (c_2^1, \dots, c_2^K, c_2^{K+1}, c_2^{K+2}, c_2^{K+3}) \\
&:= \left(g_1^{\rho_2^1}, \dots, g_K^{\rho_2^K}, g_{K+1}^{\sum_{i=1}^K \rho_2^i}, \Omega(r) \cdot \prod_{i=1}^K (y_3^i)^{\rho_2^i}, \prod_{i=1}^K (y_1^i \cdot (y_2^i)^t)^{\rho_2^i} \right),
\end{aligned}$$

with $t := H(c_2^1, \dots, c_2^{K+2})$. Observe that for $K = 1$ we obtain exactly the Cramer-Shoup PKE scheme described in the previous section.

NIZK: We use the Groth-Sahai proof system [GS08]. In order to prove knowledge of a witness (r, ρ_1, ρ_2) for $((c_1, pk_1), (hk, h, m), (c_2, pk_2)) \in L_{\mathcal{CH}}$ we use a system of multi-exponentiation equations:

$$\begin{aligned} h \cdot y^m &= g^r \\ c_1^i &= \hat{g}^{\rho_1^i} \qquad \forall i \in [K] \\ c_2^i &= g_i^{\rho_2^i} \qquad \forall i \in [K] \\ c_2^{K+1} &= g_{K+1}^{\sum_{i=1}^K \rho_2^i} \\ c_2^{K+3} &= \prod_{i=1}^K (y_1^i \cdot (y_2^i)^t)^{\rho_2^i} \\ c_1^{K+1}/c_2^{K+2} &= \hat{g}_{K+1}^{\sum_{i=1}^K \rho_1^i} \cdot \prod_{i=1}^K (y_3^i)^{-\rho_2^i}. \end{aligned}$$

This corresponds to a system of $2K+4$ equations with witness $(r, \rho_1^1, \ldots, \rho_1^K, \rho_2^1, \ldots, \rho_2^K)$, and hence using the Groth-Sahai proof system we obtain that the proof π consists of $2K+1$ commitments (each containing $K+1$ elements of $\mathbb{G}_1$) and $2K^2+4K$ proof elements (in $\mathbb{G}_2$).

Putting together the above constructions we obtain the following result.

Corollary 2. *Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be pairing based groups, and let $K \geq 1$. Under the K-Linear assumption there exists a secret-coin chameleon hash function satisfying enhanced collision resistance in the standard model, such that the hash value consists of a single group element, whereas the check value consists of $4K^2+9K+5$ group elements. In particular, the size of the check value is 18 group elements under the SXDH assumption and 39 group elements under the DLIN assumption.*

5 Integration with Bitcoin

We start by giving a short overview of Bitcoin and its main components, in Section 5.1. Section 5.2 contains a detailed explanation of how to integrate our framework within the Bitcoin infrastructure. Lastly, in Section 5.3, we report on a proof-of-concept implementation developed on top of Bitcoin core [Nak09] (initially developed by Satoshi Nakamoto).

5.1 Bitcoin Basics

Bitcoin is a peer-to-peer (P2P) electronic cash system. It has three main components, namely the P2P network among clients, the distributed consensus, and the selection of the node responsible for creating the next block. The P2P network allows Bitcoin clients to communicate reliably and distribute effectively transactions among all users.

The structure of the Bitcoin blockchain can be cast as a special case of the abstraction described in Section 3.1, where a block is represented by the tuple $B := \langle s, x, ctr \rangle$. In Bitcoin, the s value consists of the block header (minus the nonce), shown in Table 1. The value x contains all the transactions within a block, and the value ctr is the nonce value. The hash functions H and G used for the validation of the block are the SHA-256 hash function.[6]

[6]There is no technical justification of the double hashing design used in Bitcoin, but it is said that Satoshi was afraid of length-attacks on Merkle-Damgård derived hashes.

A block can be identified by its hash, which consists of the result of hashing twice the block header using SHA-256. The height of a block is its position in the blockchain starting from 0. More blocks can have the same height as forks are possible in the blockchain. The height is not stored in the header of a block. The miner will add only six fields in a block header for a total of 80 bytes (see Table 1).

The rest of the block contains a list of transactions identified by their hashes. Unlike blocks, the hash of a transaction is computed on the entire content, including everything from the header to the input and output scripts. A bitcoin transaction specifies the input transactions that have to be redeemed and the output addresses, along with input and output scripts.

Bitcoin proof of work. A proof of work is hard to compute but easy to verify. Its difficulty must be adjustable and should not be linearly computable. Proof of work must behave as a decentralized lottery. Anyone can win with a single ticket, but the probability of winning increases as the number of tickets possessed gets higher.

Bitcoin uses a proof-of-work mechanism to implement a special lottery system where one or more miners are selected essentially at random and can propose their blocks to be included in the blockchain (basically providing a probabilistic solution to a variant of the Byzantine generals problem). The nonce inside the block header is incremented until the hash of the block header returns a number less than the target. Because the target has several leading 0's, the hash must have several 0's as prefix. If the nonce is not enough, a miner can change the coinbase transaction (there is a field in there which can take arbitrary values) and generate a new Merkle root. Quite often however, miners simply change the timestamp. In the unlikely case that nothing works, it is always possible to add, remove, or change the order of transactions. Bitcoin specifies a target which we denote by D. The proof of work is successful if a miner finds a block such that the hash of its header is smaller than D.

5.2 Integrating our Framework

Our framework proposes to replace the G function (from the blockchain abstraction of Section 3.1) with a (enhanced collision resistant) chameleon hash function. In Bitcoin, this means that the inner SHA-256 function is replaced by a chameleon hash function, and its output is then used as the input to the outer SHA-256 function. The Bitcoin block header needs to be extended to accommodate the randomness of the chameleon hash (or the hash/check pair in case of a secret-coin chameleon hash), as shown in Table 1.

For the standard (immutable) Bitcoin operation, the only modifications necessary are for blocks creation and verification. There are no changes on how transactions or other aspects of the Bitcoin works. When a block is being created (i.e., mined) the miner first selects all the transactions that will be part of the block, and then it starts to fill in the block header (with the data from Table 1). Initially, the miner fills in the hash of the previous block header, the root of the Merkle tree (summarizing all transactions inside the block), the timestamp of the block creation, and the current difficulty target. Hence, it samples fresh randomness[7] and computes the chameleon hash function of the block header with the sampled randomness. The output of the chameleon hash, together with a nonce (or a counter), is used to solve the proof-of-work of the block. Once the proof-of-work is solved, the miner fills in the remaining fields of the block header, i.e., the randomness and the nonce.

After the newly created block is broadcasted, the other Bitcoin nodes can verify the validity of the block by first performing the usual verifications on the transactions, and later by

[7]For simplicity, we describe the integration using a public-coin chameleon hash. The changes necessary for a secret-coin chameleon hash are minimal, as suggested in Section 3.3.

Value	Description
Version	(4 bytes) protocol version.
Previous block	(32 bytes) the hash (twice SHA256) of the header of the previous block.
Merkle root	(32 bytes) the hash of the root of the Merkle tree that summarizes all the transactions in the block.
Timestamp	(4 bytes) approximate creation time of the block (Unix epoch).
Difficulty target	(4 bytes) difficulty target for the block.
Nonce	(4 bytes) the nonce used for the proof-of-work.
Randomness	the randomness used by the chameleon hash function. The size of this field depends on the function used.

Table 1: The Redactable Bitcoin block header.

recomputing the chameleon hash using the randomness stored in the block header, and hashing its ouput together with the nonce (via SHA-256). If the block passes all verifications, and the resulting hash is smaller than the target difficulty, then the block is considered valid and shall be added to the chain.

To manipulate the blocks we propose the creation of two new algorithms; one to remove blocks from the chain and the other to replace the contents of existing blocks. To validate the integrity of the blockchain, a verification procedure, that checks that each block header contains the hash of the previous block header, is performed on the entire blockchain. Once a block is modified or removed this integrity becomes compromised. In order to keep the integrity of the chain, a hash collision for the modified block (in the case of the redaction algorithm) or for the next block remaining on the chain (in the case of the shrink algorithm) must be computed. The hash collision algorithm (see Definition 1) computes a new randomness value that, when hashed together with the block header, outputs the same hash value as some other block (that is passed as a parameter to the function). After a collision is found, the randomness value on the block header is updated to the new computed value. As a consequence, the integrity of the blockchain is restored, and every node on the network can verify the validity of the chain.

Proof-of-Concept Implementation. We developed a proof-of-concept implementation of the redactable Bitcoin application, built on top of version 0.12 of Bitcoin core [Nak09]. The algorithms to redact a block and to shrink the chain were implemented in the centralised setting, where one special node holds the chameleon hash trapdoor key, and therefore can perform those special operations on the blockchain.[8] We implemented the chameleon hash function described in Section 3.2 using the Bignum library of OpenSSL [Ope].

Our code was developed in the same language used in Bitcoin core (C/C++). We created three main functions for the chameleon hash, namely `GenerateKey`, `ChameleonHash`, and `HashCollision`; the first function generates the public parameters and the trapdoor key, the second function takes a message and a random value and computes its hash. The last function takes an inital message and its randomness and a new message; it outputs a randomness value,

[8]For test purposes, all the parameters of the chameleon hash function, including the trapdoor key, are hard-coded in the redactable Bitcoin source code.

such that the hash of this new message and the returned randomness value is equal to the hash of the initial message and its randomness.

```
uint256 SerializeHash(const CBlockHeader& header)
{
    uint256  hash;
    uint1024 chash;
    //compute the chameleon hash of the first 76
        bytes of the block header
    ChameleonHash(header[0], 76,
        header.RandomnessR, header.RandomnessS,
        &chash);
    //now compute the SHA256 of the output of the
        chameleon hash with the nonce
    CHash256(chash, header.nNonce, &hash);
    return hash;
}
```

Figure 6: A simplified version of the overloaded SerializeHash function that computes the hash of a block header.

The first step for the integration of the chameleon hash on Bitcoin core was to extend the `CBlockHeader` class, that stores the header of a block (see Table 1). We added the members `RandomnessR` and `RandomnessS` to the class, each of size 128-bytes,[9] that holds the two randomness values required by the chameleon hash function. To modify the computation of the hash of a block header, we overloaded the function `SerializeHash` in file `hash.cpp` and created a version of this function that is called only when a block header is passed as input, in this way, when the same function is called to compute the hash of different components of the protocol, such as transactions, the correct version of the function is still called. The reason we overloaded the function `SerializeHash` and not the more obvious function `GetHash` from the `CHashWriter` class is because we must have access to the header structure before performing the hash operation, while in the `GetHash` function the block header data is already serialized. A simplified version of the code is shown in Figure 6.

Next, we modified the function `CreateNewBlock` that is called when a block is being created (file `miner.cpp`). We added the generation of fresh random values for `RandomnessR` and `RandomnessS` and stored it in the block header. In this way, when the function `SerializeHash` is called passing this block header as input, the chameleon hash is computed using the randomness values stored in the header of the block. Before having all the Bitcoin functionality operating in full, we need to create a new genesis block (described in file `chainParams.cpp`) that uses the new block header structure and the chameleon hash function. For the integration to work on an already existing blockchain a initialization procedure is necessary. This procedure would have to reconstruct all the blocks in the chain (optionally already removing and/or merging blocks) with the new block structure and using the chameleon hash function. In a production redactable Bitcoin system this initialization procedure would be run in an MPC protocol.

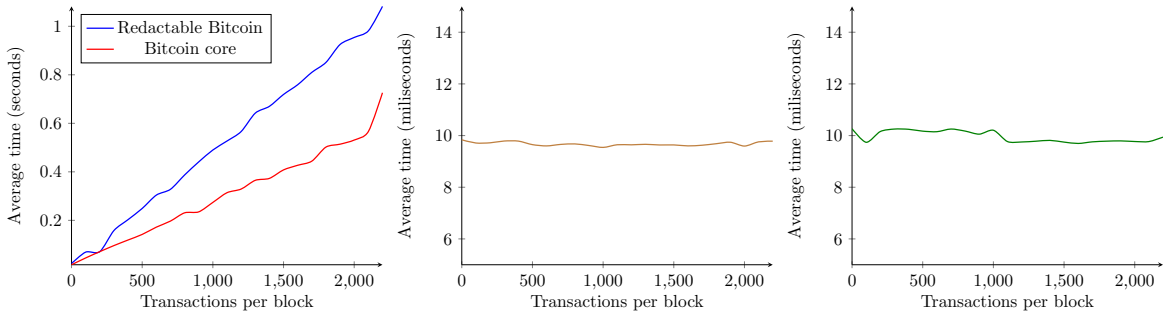
For the special operations that redact and remove blocks, we implemented the methods `RedactBlock` and `RemoveBlock` in the `CChain` class. The first method takes as input the height of the block to be redacted and the new content to put in the block, and the latter takes as input the initial and final height of a sequence of blocks to be removed.

This proof-of-concept implementation will be published as an open-source project, and its source code will be available for download soon.

5.3 Experiments

To perform the experiments (and to not pose any risk to the Bitcoin main network) we used the regression test network feature of Bitcoin core, where nodes can be simulated and connected among each other creating a private test network. We measure the time of block creation against the unmodified Bitcoin core, and the time of redacting blocks and shrinking the blockchain (removing blocks). For the measurements, we utilize python's function `time.time()` to measure

[9] The size of the randomness values (in bytes) can be significantly reduced if the chameleon hash is implemented on an elliptic curve group.



(a) Block creation. (b) Block redaction. (c) Block removal.

Figure 7: Performance evaluation of the redactable blockchain. The image on the left shows a comparison of block creation time between the redactable Bitcoin and Bitcoin core. The image on the right shows the performance of the redact block and shrink chain algorithms.

the elapsed time from the start of the the operation until the end of the operation. We note that we only measure the time to perform the operations in memory, and we disregard the time of disk access operations. The block creation experiment was performed with the block difficulty target set to 0, in this way we can precisely measure the time of the operation without the overhead of the proof-of-work. All the transactions considered in the experiment are of type `Pay-to-PubkeyHash`. The experiments were run on the hardware and software specified below.

- Intel Motherboard Server S1400SP2.
- Intel Processor Xeon E5-2430 (15M Cache,2.20 GHz, 7.20 GT/s).
- 48GB (6x8GB) RAM memory ECC 1333, DDR3.
- 2 x SSD KINGSTON SKC300S3B7A KC300 180 GB2.5 SATA III.
- 2 x 1TB HD Constellation 7200RPM.
- Ubuntu 14.04.4 LTS (GNU/Linux).
- gcc 4.9.3, Python 2.7.6.

In Figure 7a we show a comparison of the time required to create blocks in the redactable Bitcoin versus the Bitcoin core application. We note that the overhead of the redactable Bitcoin, due to the computation of a chameleon hash, is negligible and almost constant compared to Bitcoin core.

In Figure 7b and 7c we show the performance of the operations of redacting one block and removing one block from the chain, depending on the size of the block (number of transactions). The experiment was conducted on blocks of different sizes rather than on blockchains of different sizes because the running time of Algorithms 1 and 2 clearly are (almost) independent of the size of the blockchain. The only relation is due to a linear search required to find the selected block to be redacted/removed from the blockchain.

6 Conclusions

We have presented a framework to redact and compress the content of blocks in virtually any blockchain based technology. As we have argued, there are several reasons why one could prefer a redactable blockchain to an immutable one. Our approach is feasible, as implementing a redactable blockchain only requires minor modifications to the current structure of the blocks, and moreover, as our experiments showed, the overhead imposed by having a mutable blockchain is negligible.

References

[AD15] Marcin Andrychowicz and Stefan Dziembowski. Pow-based distributed cryptography with no trusted setup. In *CRYPTO*, pages 379–399, 2015.

[AdM04] Giuseppe Ateniese and Breno de Medeiros. On the key exposure problem in chameleon hashes. In *SCN*, pages 165–179, 2004.

[ADMM14a] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via Bitcoin deposits. In *Financial Crypto*, pages 105–121, 2014.

[ADMM14b] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Modeling Bitcoin contracts by timed automata. In *FORMATS*, pages 7–22, 2014.

[ADMM14c] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on Bitcoin. In *IEEE Symposium on Security and Privacy*, pages 443–458, 2014.

[ADMM15] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. On the malleability of bitcoin transactions. In *Financial Crypto*, pages 1–18, 2015.

[AFMdM14] Giuseppe Ateniese, Antonio Faonio, Bernardo Magri, and Breno de Medeiros. Certified bitcoins. In *ACNS*, pages 80–96, 2014.

[AL11] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly-secure multiparty computation. *ECCC*, 18:36, 2011.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, pages 41–55, 2004.

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, 1988.

[BCR+13] Emmanuel Bresson, Dario Catalano, Mario Di Raimondo, Dario Fiore, and Rosario Gennaro. Off-line/on-line signatures revisited: a general unifying paradigm, efficient threshold variants and experimental results. *Int. J. Inf. Sec.*, 12(6):439–465, 2013.

[BDM16] Wacław Banasik, Stefan Dziembowski, and Daniel Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. Cryptology ePrint Archive, Report 2016/451, 2016.

[Bei11] Amos Beimel. Secret-sharing schemes: A survey. In *IWCC*, pages 11–46, 2011.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *STOC*, pages 103–112, 1988.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO*, pages 421–439, 2014.

[BPRW15] Allison Bishop, Valerio Pastro, Rajmohan Rajaraman, and Daniel Wichs. Essentially optimal robust secret sharing with maximal corruptions. *IACR Cryptology ePrint Archive*, 2015:1032, 2015.

[BR14] Mihir Bellare and Todor Ristov. A characterization of chameleon hash functions and new, efficient designs. *J. Cryptology*, 27(4):799–823, 2014.

[But] Vitalik Buterin. On public and private blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/.

[Coi] Bitcoin venture capital. http://www.coindesk.com/bitcoin-venture-capital/.

[CP92] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 89–105, 1992.

[CPP06] Benoît Chevallier-Mames, Pascal Paillier, and David Pointcheval. Encoding-free ElGamal encryption without random oracles. In *PKC*, pages 91–104, 2006.

[CS98] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO*, pages 13–25, 1998.

[CTZD10] Xiaofeng Chen, Haibo Tian, Fangguo Zhang, and Yong Ding. Comments and improvements on key-exposure free chameleon hashing based on factoring. In *Inscrypt*, pages 415–426, 2010.

[CZK04] Xiaofeng Chen, Fangguo Zhang, and Kwangjo Kim. Chameleon hashing without key exposure. In *ISC*, pages 87–98, 2004.

[CZS+10] Xiaofeng Chen, Fangguo Zhang, Willy Susilo, Haibo Tian, Jin Li, and Kwangjo Kim. Identity-based chameleon hash scheme without key exposure. In *ACISP*, pages 200–215, 2010.

[CZS+14] Xiaofeng Chen, Fangguo Zhang, Willy Susilo, Haibo Tian, Jin Li, and Kwangjo Kim. Identity-based chameleon hashing and signatures without key exposure. *Inf. Sci.*, 265:198–210, 2014.

[CZT+11] Xiaofeng Chen, Fangguo Zhang, Haibo Tian, Baodian Wei, and Kwangjo Kim. Discrete logarithm based chameleon hashing and signatures without key exposure. *Computers & Electrical Engineering*, 37(4):614–623, 2011.

[Dam87] Ivan Damgård. Collision free hash functions and public key signature schemes. In *EUROCRYPT*, pages 203–216, 1987.

[DeR] Chris DeRose. Why blockchain immutability is a perpetual motion claim. http://www.coindesk.com/immutability-extraordinary-goals-blockchain-industry/?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed:+CoinDesk+%28CoinDesk+-+The+Voice+of+Digital+Currency%29.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.

[DHLW10] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana López-Alt, and Daniel Wichs. Efficient public-key cryptography in the presence of key leakage. In *ASIACRYPT*, pages 613–631, 2010.

[EGM96] Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. *J. Cryptology*, 9(1):35–67, 1996.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, 31(4):469–472, 1985.

[Eth] Ethereum project. https://www.ethereum.org/.

[FF02] Marc Fischlin and Roger Fischlin. The representation problem based on factoring. In *CT-RSA*, pages 96–113, 2002.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *CRYPTO*, pages 152–168, 2005.

[FKMV12] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. On the non-malleability of the Fiat-Shamir transform. In *INDOCRYPT*, pages 60–79, 2012.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310, 2015.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001.

[GQ88] Louis C. Guillou and Jean-Jacques Quisquater. A "paradoxical" indentity-based signature scheme resulting from zero-knowledge. In *CRYPTO*, pages 216–231, 1988.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In *ASIACRYPT*, pages 444–459, 2006.

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In *EUROCRYPT*, pages 415–432, 2008.

[HC] Steve Hargreaves and Stacy Cowley. How porn links and ben bernanke snuck into bitcoin's code. http://money.cnn.com/2013/05/02/technology/security/bitcoin-porn/index.html.

[HK07] Dennis Hofheinz and Eike Kiltz. Secure hybrid encryption from weakened key encapsulation. In *CRYPTO*, pages 553–571, 2007.

[Hop] Curt Hopkins. If you own Bitcoin, you also own links to child porn. http://www.dailydot.com/business/bitcoin-child-porn-transaction-code/.

[Inf] Blockchain Info. Bitcoin hashrate distribution. https://blockchain.info/pools.

[KMB15] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *CCS*, pages 195–206, 2015.

[KMS+15] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. *IACR Cryptology ePrint Archive*, 2015:675, 2015.

[KR00] Hugo Krawczyk and Tal Rabin. Chameleon signatures. In *NDSS*, 2000.

[KT15] Aggelos Kiayias and Qiang Tang. Traitor deterring schemes: Using bitcoin as collateral for digital content. In *CCS*, pages 231–242, 2015.

[Man11] Alfonso Cevallos Manzano. Reducing the share size in robust secret sharing. Master's thesis, Mathematisch Instituut Universiteit Leiden, the Netherlands, 2011.

[Nak09] Satoshi Nakamoto. Bitcoin core. https://github.com/bitcoin/bitcoin, 2009.

[NR94] Kaisa Nyberg and Rainer A. Rueppel. Message recovery for signature schemes based on the discrete logarithm problem. In *EUROCRYPT*, pages 182–193, 1994.

[NT01] Moni Naor and Vanessa Teague. Anti-presistence: history independent data structures. In *ACM STOC*, pages 492–501, 2001.

[Oka92] Tatsuaki Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *CRYPTO*, pages 31–53, 1992.

[Ope] OpenSSL project. http://www.openssl.org/.

[OS90] H. Ong and Claus-Peter Schnorr. Fast signature generation with a Fiat Shamir-like scheme. In *EUROCRYPT*, pages 432–440, 1990.

[PB] Kevin Petrasic and Matthew Bornfreund. Beyond bitcoin: The blockchain revolution in financial services. http://www.whitecase.com/publications/insight/beyond-bitcoin-blockchain-revolution-financial-services.

[Pea] Jordan Pearson. The bitcoin blockchain could be used to spread malware, interpol says. http://motherboard.vice.com/read/the-bitcoin-blockchain-could-be-used-to-spread-malware-interpol-says.

[PS15] Rafael Pass and Abhi Shelat. Micropayments for decentralized currencies. In *CCS*, pages 207–218, 2015.

[PSas16] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. Cryptology ePrint Archive, Report 2016/454, 2016.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *STOC*, pages 73–85, 1989.

[RB07] Phillip Rogaway and Mihir Bellare. Robust computational secret sharing and a unified account of classical secret-sharing goals. In *CCS*, pages 172–184, 2007.

[RKS15] Tim Ruffing, Aniket Kate, and Dominique Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of Bitcoins. In *CCS*, pages 219–230, 2015.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.

[SCO+01] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In *CRYPTO*, pages 566–598, 2001.

[Sco02] Mike Scott. Authenticated ID-based key exchange and remote log-in with simple token and PIN number. *IACR Cryptology ePrint Archive*, 2002:164, 2002.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Sha07] Hovav Shacham. A Cramer-Shoup encryption scheme from the linear assumption and from progressively weaker linear variants. *IACR Cryptology ePrint Archive*, 2007:74, 2007.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT*, pages 256–266, 1997.

[ST01] Adi Shamir and Yael Tauman. Improved online/offline signature schemes. In *CRYPTO*, pages 355–367, 2001.

[Ten] Jeni Tennison. What is the impact of blockchains on privacy? https://theodi.org/blog/impact-of-blockchains-on-privacy.

[U.S13] U.S Government Accountability Office. Financial regulatory reform: Financial crisis losses and potential impacts of the dodd-frank act. Technical report, 2013.

[WB86] Lloyd R. Welch and Elwyn R. Berlekamp. Error correction of algebraic block codes, 1986. US Patent 4,633,470.