On Bitcoin Security in the Presence of Broken Crypto Primitives

August 15, 2016

Ilias Giechaskiel
University of Oxford
Oxford, United Kingdom
ilias.giechaskiel@cs.ox.ac.uk

Cas Cremers
University of Oxford
Oxford, United Kingdom
cas.cremers@cs.ox.ac.uk

Kasper B. Rasmussen
University of Oxford
Oxford, United Kingdom
kasper.rasmussen@cs.ox.ac.uk

Abstract

Digital currencies like Bitcoin rely on cryptographic primitives to operate. However, past experience shows that cryptographic primitives do not last forever: increased computational power and advanced cryptanalysis cause primitives to break frequently, and motivate the development of new ones. It is therefore crucial for maintaining trust in a crypto currency to anticipate such breakage.

We present the first systematic analysis of the effect of broken primitives on Bitcoin. We identify the core cryptographic building blocks and analyze the various ways in which they can break, and the subsequent effect on the main Bitcoin security guarantees. Our analysis reveals a wide range of possible effects depending on the primitive and type of breakage, ranging from minor privacy violations to a complete breakdown of the currency.

Our results lead to several observations on, and suggestions for, the Bitcoin migration plans in case of broken cryptographic primitives.

1 Introduction

Cryptocurrencies such as Bitcoin rely on cryptographic primitives for their guarantees and correct operation. However, cryptographic primitives typically get broken or weakened over time. This is due to progress in cryptanalysis as well as advances in the computational power of the attackers. For example, for a timeline overview of breakages of hash functions, see [4]. It is therefore prudent to expect that in time, the cryptographic primitives used by Bitcoin will be partially, if not completely, broken.

In anticipation of such breakage, the Bitcoin community has created a wiki page that contains draft contingency plans (see Appendix C). The focus of these is on “getting the word out” and “coordination”. The wiki describes a few scenarios in which these plans will be necessary, including “broken” SHA256 and ECDSA algorithms. However, what exactly is meant by “broken” is not fully explained. Moreover, the subsequent steps after a contingency are hand-wavy and incomplete, e.g., “once the plans themselves are well-accepted, code implementing the plans can be written and tested in case the code is ever required” [11]. To the best of our knowledge, no adequate mechanism has been built into Bitcoin, and no plans for partial breakage (or weakening of a primitive) have been considered.

In practice, the situation is not black-and-white. Instead of abruptly breaking completely, cryptographic primitives usually break gradually. With hash functions, for example, it is common that first a single collision is found. This is then later generalized to multiple collisions, and only later do arbitrary collisions become feasible to compute. In parallel, the complexity of attacks (such as collisions) decreases to less-than-brute-force, and computational power increases. Finally, quantum computing will make some attacks easier, e.g., Grover’s pre-image attack [23], or Shor’s algorithm for discrete log computation [45].

Even if such attacks are years away from being practical, it is crucial to anticipate the impact of broken primitives, so that appropriate and effective contingency plans can be put in place. Our work contributes towards filling this gap.

Contributions We consider our main contributions to be the following. First, we provide the first systematic analysis of the impact of broken primitives on Bitcoin. While the Bitcoin community has considered some high-level cases, we perform a fine-grained systematic analysis. We do this by identifying the core cryptographic building blocks within Bitcoin and consider the various ways in which such primitives are broken in practice. We analyze both the failure of individual primitive properties, and combinations of broken primitives, and highlight their impact on the main security properties of Bitcoin.

Second, we describe in detail the range of consequences different breaks have. For example, one interesting and counter-intuitive result is that an adversary with access to a simple pre-image oracle does not gain

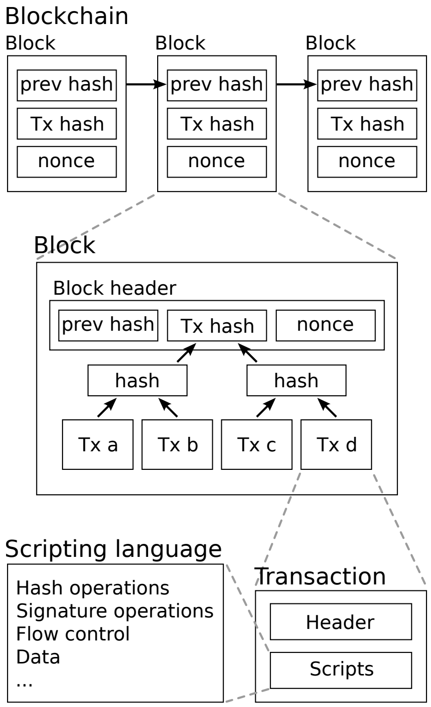

Figure 1: The blockchain data structure. This forms the basis of the public, append-only ledger where all transactions are recorded.

an advantage for mining by just working on the block header. However, with a few calls, the adversary can use the flexibility of the coinbase transaction and succeed in mining the next block with high probability.

Third, our investigations raise concerns about the currently specified migration plans for Bitcoin. Although the current plans are a first step in the right direction, we show that they are both overly conservative in some respects, as well as inadequate in others. For example, it is not clear that if someone owns a Bitcoin now, they will still have access to it in a year, if a primitive has broken in the meantime.

As a tangential fourth contribution, we introduce an oracle model for hash functions that unifies several existing types of breakage and allows us to specify forms of hash function breakage that are closer to real-world attacks.

We proceed as follows: after introducing the relevant background (Section 2), we propose our system and adversary model (Section 3). Then, we analyze the effects of broken primitives on the design of Bitcoin. In particular, we consider the hashing building blocks (Section 4) and signature schemes (Section 5), before considering combinations of primitive breaks (Section 6). We then revisit the current implementation of Bitcoin and its contingency plans (Section 7). After discussing related work (Section 8), we conclude (Section 9).

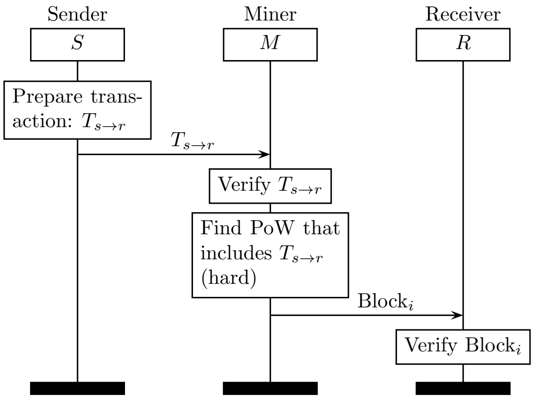

Figure 2: A high-level view of a Bitcoin transaction.

2 Background

In this section, we give a description of Bitcoin, the popular peer-to-peer (P2P) cryptocurrency introduced in 2008 by Satoshi Nakamoto [39]. Figure 1 shows a high-level view of the main component of Bitcoin—the blockchain—which will guide this section. The blockchain is a public log of all Bitcoin transactions that have occurred, combined together in components called blocks. Transactions use a scripting language that determines the owners of coins (Section 2.1), and it is up to miners to ensure that only valid transactions occur. To ensure that nobody can change or remove past transactions, miners have to solve a hard computational puzzle, known as a Proof-of-Work (Section 2.2), so from a user's perspective, the process of making a transaction resembles Figure 2. The final component of Bitcoin is its underlying P2P network which enables the distributed communication (Section 2.3). We do not introduce or analyze components which are outside of the main protocol, such as wallets.

2.1 Transactions and Scripts

Bitcoin is an electronic cash system [39], so *transactions* to transfer coins between different users are central to its structure. Coins in Bitcoin are essentially unspent transaction outputs, whose unit is a "satoshi", equal to 10^{-8} Bitcoins or BTCs. A transaction, then, can be thought of as a list of inputs – unspent transactions in the blockchain – and a list of outputs – addresses to which to transfer the coins. In order to ensure that only the owner can spend his coins, each input and output is accompanied by a *script*. For outputs, this "locking" script contains the conditions under which the output can be redeemed (*scriptPubKey*), while for inputs, an "unlocking" script contains proof in the form of a cryptographic signature (*scriptSig*) that these conditions have been met.

Figure 3 contains a high-level example of how a trans-

```
input  : source outputs and destination addresses and values
output : Bitcoin transaction
/* Specify where the Bitcoins of this
   transaction are coming from                    */
foreach source output $o_n$ do
    pub, priv ← Get key pair for $o_n$
    sig ← $\mathtt{Sign}_{\mathsf{priv}}$(selected fields) /* See [44]    */
    ScriptSig ← sig pub
    Add ($o_n$, ScriptSig) to list of inputs
end
/* Specify one or more addresses to receive
   the Bitcoins                                   */
foreach destination address addr do
    ScriptPubKey ← DUP HASH160 addr EQV CHKSIG
    Add (value, ScriptPubKey) to list of outputs
end
Transaction ← (version, inputs, outputs, lock-time)
```

Figure 3: Procedure for creating a Pay-to-Public-Key-Hash transaction with multiple inputs and outputs.

action is constructed. The public keys of receivers are converted into Pay-to-Public-Key-Hash output scripts (explained below), while the input scripts are outputs of previous transactions and are accompanied by signatures of the corresponding public keys by senders.

The *scripts* mentioned above are sequences of instructions called *opcodes* that get executed by special nodes called miners. Bitcoin's *scripting language* is stack-based, and without loops. There are opcodes for string, arithmetic, and cryptographic operations, as well as for representing data, controlling the program flow and the stack. However, many opcodes are disabled to prevent Denial-of-Service (DoS) attacks which exploit computationally intensive opcodes. Most nodes thus only accept and relay a certain set of *standard scripts* of fixed format, which are presented below.

- *Public-Key* The unlocking script must sign the transaction under the provided key. The corresponding scriptPubkey is `<pubkey> OP_CHECKSIG`, with scriptSig equal to `<signature>`.
- *Pay-to-Public-Key-Hash (P2PKH)* The unlocking script must provide a public key which hashes to the given value, and must then sign the transaction under this key. The corresponding scriptPubkey is `OP_DUP OP_HASH160 <pubkeyHash> OP_EQUALVERIFY OP_CHECKSIG`, with scriptSig equal to `<signature> <pubkey>`.
- *Multi-Signature* An M-of-N ($N \leq 15$) multi-signature scheme provides N public keys, and requires M signatures in the unlocking script. In this case, scriptPubkey is `M <pubkey 1> ...<pubkey N> N OP_CHECKMULTISIG`, with scriptSig being `OP_0 <signature 1> ...<signature M>`.

```
input  : Bitcoin transaction
output : valid or invalid
/* Verify each input to the transaction           */
foreach transaction input do
    SrcOut ← Get source output
    ScriptPubKey ← from SrcOut
    ScriptSig ← from current input
    $\mathtt{Verify}_{\mathsf{pub}}$(ScriptSig || ScriptPubKey)
end
```

Figure 4: Procedure to verify a transaction's use of cryptographic primitives. The full procedure is in Appendix A.

- *Pay-to-Script Hash (P2SH)* This script is the hash of a non-P2SH standard transaction. The unlocking script then provides the full script (which hashes to this value) as well as any necessary signatures. This script is typically used to shorten the length of multi-signature transactions. The scriptPubkey field in this case is `OP_HASH160 <scriptHash> OP_EQUAL`, while scriptSig is `<signatures> <script>`.
- *Data Output (OP_RETURN)* The output cannot be redeemed, but can be used to store up to 40 arbitrary bytes such as human-readable messages in the blockchain for a fee: scriptPubkey is `OP_RETURN <data>` and there is no corresponding scriptSig.

Every transaction needs to be valid: it needs to contain all the required fields, all signatures must be valid, and the output scripts must be one of the standard ones. This is a task that miners undertake for a small fee. Though some non-standard scripts can be accepted by some miners for a higher fee, we do not cover these in our analysis. The relevant parts of validating a transaction from a cryptographic perspective are summarized in Figure 4.

2.2 Mining and Consensus

To ensure that no coin is used more than once, every transaction is made public, through a global, append-only ledger called the *blockchain*, consisting of *blocks* combining transactions in a Merkle Tree [38]. New blocks become a part of the blockchain through a process called *mining*: miners need to find a value (nonce) such that the hash of a block's header is less than a given target $h(hdr||nonce) < T$. The idea behind this *proof-of-work* scheme is that the probability of creating the next block is proportional to the miner's computational power, and because miners receive transaction fees (as well as a block creation reward through special "coinbase" transactions) they are incentivized to do the work, which includes validating transactions and blocks (Figure 5).

Due to the probabilistic nature of mining, the presence of adversaries, and networking delays, miners may

```
input  : Bitcoin block
output : valid or invalid
/* Verify block header                                  */
Verify $\mathsf{Hash}(\textit{block header}) < \mathsf{target}$
Verify Merkle hash
Verify $\mathsf{Hash}(\textit{prev block}) = \mathsf{prev\_hash}$
/* Verify each transaction input in block               */
foreach transaction input in the block do
    Check that referenced output transaction exists and hasn't
      already been spent
    Verify signatures
end
```

Figure 5: Procedure to verify a block's cryptographic primitives. For the full procedure see Appendix B.

disagree on the current state of the blockchain. This is known as a *fork*. To deal with this issue, there are hardcoded blocks included in the clients, known as *checkpoints*, starting from the first block, called the *genesis block*. In addition, honest (non-adversarial) miners work on the longest blockchain they become aware of, when other nodes announce new blocks and transactions. This way, nodes eventually reach consensus [13] [20].

These temporary forks enable *double spending*: an adversary can have different transactions in different branches of the fork using the same inputs but different outputs. However, because the probability of "deep" forks where branches differ in the top N blocks drops exponentially in N, receivers usually wait for multiple confirmation blocks ($N = 6$ is typical).

If a miner or a group of collaborating miners (called a *pool*) is in control of a high enough proportion of the total computational power (51% [34], or even less [19]), then they can possibly destabilize the system.

2.3 Network

The last key component of Bitcoin is its Peer-to-Peer (P2P) network that allows it to operate in a distributed fashion. Transactions and blocks are broadcast by participating nodes to their peers, and then relayed further to flood the network if they meet the relay policies in place to prevent DoS attacks. Not every node in the P2P network is a miner, and potentially not every node has access to the full chain. Though "full" clients download the entire blockchain through `getblocks` messages, "lightweight" clients that use Simple Payment Verification (SPV) only download headers and the relevant transactions (with the corresponding Merkle Trees). For receivers, it is important to choose one's networking peers so that senders must broadcast the transaction to the entire network instead of just to them [50], while connecting to multiple random nodes is important to ensure that transactions are not hidden by dishonest nodes [3].

2.4 Upgrading the Protocol

Because of all the verification steps involved, it is important for participating nodes to be in agreement as to what the precise transaction verification algorithm is. However, to fix bugs or to introduce extensions, the protocol itself may need to change. Since not all nodes upgrade at the same time, a protocol change may introduce forks. If the validation rules in the upgrade become stricter, then the protocol remains backwards-compatible, so nodes running the old software will accept transactions of the new type. This is a *softfork*. A *hardfork*, on the other hand, is not backwards-compatible, and thus requires the entire network to upgrade, as old software would reject new transactions and blocks as invalid.

The distinction between the two fork types is important for Bitcoin's future (Section 7), since when there is disagreement in the community, a transition becomes much harder, as happened recently over the size of blocks [24].

3 System and Adversary Model

In this section we describe our model for Bitcoin and discuss the goals and powers that an adversary might have in the presence of broken cryptographic primitives.

In our model, we distinguish between 4 entities: senders, receivers, miners, and networking nodes. Senders and receivers, collectively referred to as *users*, wish to exchange Bitcoins via transactions, but do not care about the details of the underlying system. They wish to send payments and receive incoming deposits, and care about the amount of money under their control.

Transactions are transmitted via the underlying P2P network. Miners have their own (possibly different) copy of the blockchain, and have different hashing capacities. For our model, we consider pools as single miners with a large hashing capability.

We distinguish between two roles for the adversary: that of a simple user, and that of a miner. As a user, the adversary takes the role of a sender, and aims to make money, either by successfully double spending or by spending from another user's wallet.

As a miner, the adversary controls a proportion α of the mining power, that is not normally sufficient to take over Bitcoin (i.e., $\alpha < 0.5$). The adversary is also assumed to be in control of a proportion β of the nodes in the P2P network, so that he can attempt to split the network temporarily in the presence of a suitable vulnerability, but cannot be confident that such attempt will succeed.

The economic aspects of Bitcoin are considered out of scope, and we also do not consider developers as a threat.

Finally, we do not investigate adversarial attacks of an individual miner against his own pool, thus allowing us to consider pools as single entities of more mining power.

4 Broken Hashing Primitives

In this section, we look at the cryptographic hash functions in Bitcoin, and analyze the effect of a break in one of the properties of first and second pre-image and collision resistance. We generalize these into a single property called chosen-format bounded pre-image resistance.

4.1 Hashing in Bitcoin

In the original Bitcoin paper [39], the concrete primitives used are not specified: there were no "addresses" but just public keys, and the hash used for mining and the Merkle tree was just referred to as a hash function. The current Bitcoin implementation, going back to at least version 0.1.0 [40] uses two hash functions.

Main Hash This hash function occurs frequently in Bitcoin. It has an output of 256 bits and requires applying SHA256 twice: $H_M(x) = \text{SHA256}(\text{SHA256}(x))$. It is the hash used for *mining* (Proof-of-Work): miners need to find a nonce such that the double SHA256 hash of a block header is less than a "target" hash. It is also used to hash transactions within a block into a *Merkle Tree*, a structure which summarizes the transactions present within a block. Because Merkle Trees allow efficient verification of whether a given transaction is present in a given block, they are used extensively in "light-weight" clients that only perform *Simplified Payment Verification (SPV)* and thus do not download full blocks [3]. Finally, it is the hash used for transactions signed with a user's private key (see [44] for details).

Address Hash The second hash function is used as part of the Pay-to-Public-Key-Hash (P2PKH) and the Pay-to-Script-Hash (P2SH) scripts. Its output is 160 bits, and it is concretely instantiated as $H_A(x) = \text{RIPEMD160}(\text{SHA256}(x))$.

4.2 Modeling Hash Breakage

In this subsection we analyze how hashes break in terms of their building blocks, and introduce our oracle model for their breakage.

4.2.1 Identifying Hashing Building Blocks

Though we have introduced the two hashes, H_A and H_M as single entities, they are built out of two well-known, standard hash functions: RIPEMD160 and SHA256. Because within Bitcoin, neither RIPEMD160 nor SHA256 appear individually, we consider attacks against their combinations of H_A and H_M. That way our arguments can be extended for any future design, regardless of the concrete instantiation of H_A and H_M.

To relate the attacks we discover back to the concrete primitives in Section 7, we now show that for collisions and second pre-images, only one of the two nested hashes needs to be broken, while for pre-images both need to be broken. For a cryptographic hash function $h(x)$, these three properties are defined as follows:

1. *Pre-image resistance* Given y it is hard to find x with $h(x) = y$.
2. *Second pre-image resistance* Given x_1, it is hard to find $x_2 \neq x_1$ with $h(x_1) = h(x_2)$.
3. *Collision resistance* It is hard to find distinct $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$.

where "hard" refers to computational infeasibility, since hash functions have a fixed-length output, so collisions always exist. We investigate each of these properties, when $h = h_1 \circ h_2$ is a composition of two hash functions.

Pre-image resistance h is broken only when both h_1, and h_2 are broken. In one direction, assume that we have a pre-image algorithm for h, that returns x on input y. Then, to find a pre-image for y under h_2, run the algorithm on $h_1(y)$ for output x. If $h_2(x) = y$, then x is a pre-image for y under h_2. Else $h_2(x) \neq y$ and $(h_2(x), y)$ forms a collision (or second pre-image) for h_1. Conversely, if there is an algorithm for both h_1 and h_2 pre-images, then to get a pre-image of y under h, one finds a pre-image x_1 of y under h_1, and then a pre-image x_2 of x_1 under h_2. x_2 is then a pre-image of y under h.

Second pre-image resistance h is only as strong as the inner function h_2. In one direction, assume that given x_1 one can find $x_2 \neq x_1$ such that $h_2(x_1) = h_2(x_2)$. Then clearly $h(x_1) = h(x_2)$.[1] In the other direction, assuming that given x_1, one can find $x_2 \neq x_1$ such that $h(x_1) = h(x_2)$, then either $h_2(x_1) = h_2(x_2)$ for a second pre-image attack on h_2 or $h_2(x_1) \neq h_2(x_2)$ for a collision (and second pre-image of $h_2(x_1)$) on h_1.

Collision resistance h is again only as strong as h_2. A collision (x_1, x_2) for h_2 is clearly a collision for h, and a collision (x_1, x_2) for h is either a collision for h_2 or $(h_2(x_1), h_2(x_2))$ is a collision for h_1.

4.2.2 Modeling Hash Breakage Variants

Though the three properties of first and second pre-image and collision resistance are typically used to describe hashes, they don't accurately capture all types of breakages that may be beneficial to a malicious miner and other

[1] The same can be said if h_1 is vulnerable to second pre-image attacks and h_2 is vulnerable to first pre-image attacks.

adversaries. A breakage in the real world typically exploits the concrete internal structure of the hash function, and as a result the attacks involved may be more powerful than a simple collision or pre-image. An adversary might thus have more control over the structure of the pre-image or the target value. For example, mining expects the hash to be smaller than a given target, a property which cannot be expressed using traditional pre-image oracles, as we explain in detail in Section 4.3.

For this reason, we introduce a more general oracle model to enable our analysis. We first describe the format of the oracle we choose to use, and then go over the details of why it has the specified format, and how it also generalizes the traditional types of primitive breakage using appropriate parameters.

We call the oracle a *chosen-format bounded pre-image oracle P*, which on input (a, b, y_l, y_h, i) returns an x_i such that $y_l \leq h(a||x_i||b) \leq y_h$ or $\perp$ if none exists. In other words, the oracle returns a value $X_i = a||x_i||b$ such that its beginning and end are caller-supplied, and such that the its hash is within a given target range. Moreover, the oracle is deterministic such that the same x_i is returned each time and $x_i \neq x_j$ for $i \neq j$ and if given an optional parameter s, the returned x_i has size s bits. That is to say, the oracle can be called multiple times to get different pre-images, and the user is also able to specify the length of the pre-image in bits.

We now explain the motivation behind these parameters. First of all, being able to specify a, b, and the length of the input allows us to have control over the format of the pre-image. This is important, because in our discussion we want to speak of pre-images and collisions that follow the format of transactions and block headers. Note that fixing a is not uncommon, as it can be used to describe chosen-prefix collisions. Hash functions are typically not symmetric, so fixing b is usually harder than fixing a, but it is not necessary for most attacks we describe, and even when it is, we still wish to enable the adversary to control the format of the hashed value. This is similar to the Chosen-Target-Forced-Midfix attack by Andreeva and Mennink [1].

Using bounds on the target range is necessary to describe some attacks against the proof-of-work (PoW) scheme. Even though PoW only requires the upper bound y_h, we include y_l for symmetry, and because it is used to implement a traditional collision attack as we see below.

The oracle needs an index parameter for the following reason. When there is no length restriction on the pre-image, there are potentially infinitely many pre-images, even for a well-designed hash function when each output is equally likely. When the input must have size s bits, there are 2^s possible pre-images, which is exponential. Thus, the oracle cannot return the entire set of pre-images to a polynomially-bounded adversary. This is why we desire $x_i \neq x_j$ for $i \neq j$: we want to be able to access as many distinct pre-images as we want. Finally, we desire the returned values on different indices to be distinct, and we want there to be no "gaps", i.e., if the oracle returns $\perp$ on i it should also return $\perp$ on $i+1$, so that the adversary can stop querying the oracle after receiving a $\perp$.

We show that our oracle model is a generalization of the traditional breakages by showing how an adversary with access to our oracle can break the three hash properties.

Pre-image Getting a pre-image of y amounts to calling P on $(\perp, \perp, y, y, 0)$, so the adversary can break pre-image resistance with a single call to the oracle.

Second pre-image Getting a second pre-image given x is almost identical, but potentially requires two oracle calls: call P on $(\perp, \perp, h(x), h(x), 0)$, and if that returns x, call P on $(\perp, \perp, h(x), h(x), 1)$.

Collision Getting a collision is not as straightforward. Let $h : \{0,1\}^* \rightarrow \{0,1\}^n$ be the hash function in question. First of all, it is not always the case that every $y \in \{0,1\}^n$ has a pre-image (let alone two), even though probabilistically this holds true for a well-designed hash function. For instance, consider h', where $h'(x) = 1$ when $h(x) = 0$, and $h'(x) = h(x)$ otherwise. Then, h' is strong if h is strong, but does not hit 0. However, by exploiting the pigeonhole principle and binary search, one can make $\lg(n)$ calls to the oracle to generate a collision.

The idea is to call P on $(\perp, \perp, y_l, y_h, y_h - y_l + 2)$. If the oracle returns anything but $\perp$, there are more pre-images than possible hashes within the range $[y_l, y_h]$. Then, one can perform a binary search with initial $y_l = 0^n$, $y_h = 1^n$ to determine a value y that has at least 2 pre-images.

Chosen-prefix collision To get a chosen-prefix collision, i.e. given p find two values $x \neq x'$ such that $h(p||x) = h(p||x')$, one performs the same procedure as for getting a normal collision, but with $a = p$.

For the rest of our analysis, we still use the notions of pre-images and collisions, unless we need the additional power of the more powerful oracle. We summarize our results in Table 1.

4.3 Main Hash

In this section we analyze the main hash H_M, which is used for mining, in Merkle Trees, and with signatures. We discuss all three separately.

4.3.1 Mining

Mining (Pre-Image against Fixed Merkle Root) Miners search for block headers whose hash is below some target value. We analyze the probability that an adversary with access to a pre-image oracle can break mining. Because some fields in the header are fixed as explained

Breakage	**Address Hash** (H_A)	**Main Hash** (H_M)
Collision	Repudiate payment	Destroy coins
Second pre-image	Repudiate payment	Double spend and steal coins
Pre-image	Uncover address	Complete failure of the blockchain ($2n$ calls)
Bounded pre-image	All of the above	Complete failure of the blockchain (n calls)

Table 1: Summary of the effects on Bitcoin for different types of breakage in the two hash functions used.

below, we assume that the oracle allows the caller to specify the pre-image prefix.

To simplify the analysis, we assume that the target hash of length n needs to start with d zeros. In reality, the target can be higher by at most a factor of 2, so the assumption introduces up to 1 bit of extra work.

If the adversary controls $b \leq n$ bits of the input, there are 2^b possible inputs to the hash function. These need to map to one of the 2^{n-d} values in the range $[0, \text{target})$, and will be uniformly distributed across 2^n values. This gives the expected number of b-bit pre-images as

$$E[\text{\# pre-images}] = 2^b \cdot \frac{2^{n-d}}{2^n} = 2^{b-d}$$

The adversary can only query the pre-image oracle for specific target hashes. Because there are 2^{b-d} b-bit pre-images, distributed across the 2^{n-d} values, the probability that a given hash in $[0, \text{target})$ has a b-bit pre-image is:

$$P[\text{correct pre-image}] = \frac{2^{b-d}}{2^{n-d}} = 2^{b-n}$$

This probability does not depend on d, as one might expect. This is because by increasing d to reduce the number of valid hashes, the adversary also reduces the expected number of b-bit pre-images. Assuming the adversary is allowed 2^a queries to the oracle, the probability of breaking mining becomes

$$P[\text{success}] = 2^a \cdot 2^{b-n} = 2^{a+b-n}$$

To calculate b, we explore all fields in the block header. The version number (`nVersion`), as well as the hashes of the previous block header (`hashPrevBlock`), and of the current Merkle root hash (`hashMerkleRoot`) are fixed. However, the adversary has partial control over the remaining fields in the header. For the timestamp field (`nTime`), the value can be within 7200 seconds of the current median/average, giving the adversary approximately 13 bits of freedom. Moreover, the adversary has complete control over the 32 bits of the nonce (`nNonce`).

The `nBits` field describing the target difficulty is a bit more complicated to explain. The 4 bytes $0xAABBCCDD$ describe the number $0xBBCCDD \cdot 256^{0xAA-3}$, and the protocol only checks that the produced number is at most the target value given by the consensus. At the time of writing, this value is $0x180928f0$, meaning that the adversary can use approximately 28 of the 32 bits.

All the fields combined give $b = 73$. With $n = 256$, and if we allow 2^{80} calls to the oracle, i.e., $a = 80$, the probability of success is only $2^{80+73-256} = 2^{-103}$, which is still negligible.

Mining (Pre-Image against Variable Merkle Root) In the above derivation we fixed the root hash of the current transactions, though an adversary has partial control over them as well. One approach would be to selectively exclude and/or reorder transactions, but the effect is the same as fixing a different root hash. The alternative is to work backwards: start with a fixed hash, and include the 32 bytes of the Merkle hash in b, working backwards to reconstruct the tree of transactions. However, even with semantically valid transactions, there is negligible probability that they correspond to valid and unspent outputs in the blockchain, or have valid signatures.

However, Bitcoin does not enforce a minimum number of transactions in a block, and the coinbase transaction which generates new coins has a variable-length input that is controlled by miners: coinbase transactions have the same fields as regular transactions, but they only have one transaction input whose script is not fixed, but can contain arbitrary data of up to 100 bytes (see [44]).

As a result, a malicious miner with access to an oracle that can take as input the prefix and the suffix of the pre-image can do the following:

1. Pick an arbitrary target T and get a pre-image for $h(a||x||b) = T$ where the desired x is the `hashMerkleRoot` field, and a, b are the remaining fields in a block header. Because the root is 256 bits, there is a pre-image with high-probability, but if not, repeat with some other random target T'.
2. Pick a length l for the script, and fix all other fields for the coinbase transaction. Solve $h(a'||y||b') = T$ where a', b' are the remaining fields for the coinbase transaction. Because the number of free bytes is up to 100, the process succeeds with high probability.

Mining (Bounded Pre-Image) An adversary with access to our chosen-format, bounded pre-image oracle P can simply call P on $(hdr, \perp, 0^n, y_t, 0, s)$, where y_t is the target hash, hdr is the beginning of the block header, $n = 256$ is the size of the output, and $s = 32$ is the size of the required nonce.

Breaking mining may result in completely breaking the security of the blockchain since an adversary has a much higher probability of creating deep forks, thus reversing transactions or double spending. We note that mining using a pre-image oracle requires twice as many calls compared to mining using the bounded pre-image oracle.

Mining (Collisions, Second Pre-Images) Collisions and second pre-images are only useful for mining if the pre-images start with d zeros. Assuming the pre-images contain valid transactions and signatures (negligible probability), a miner can fork the chain.

4.3.2 Merkle Trees

Altering existing blocks By repeating the argument given for mining, we see that an adversary cannot find a valid second pre-image of an entire block except with negligible probability. Pre-images do not give the adversary new information, as they already accompany the hash value. Collisions are also not useful, as both values are controlled by the attacker, so cannot alter existing blocks.

Attacking new blocks For new blocks, an adversary with sufficient network control can use a collision or second pre-image to split the network, even with an invalid block. By transmitting a block with the same hash as the new valid one, the adversary can cause miners to reject both blocks, and possibly reverse the transactions it contained. This is not a strong assumption: in July 2015, some miners generated invalid blocks, and some clients did not detect these blocks as invalid, a situation which matches the problem identified above [9].

The same attack works for invalidating new transactions. By creating a collision or second pre-image, the attacker can create and transmit two conflicting transactions to split the network and double-spend. This can be used to fool a vendor: one transaction – transmitted to just the vendor – appears to be transferring money to him, while the other transaction – transmitted to the rest of the network – contains the collision which transfers funds elsewhere. Pre-images are again not relevant to an attacker, as they always accompany the hashed value.

4.3.3 Main Hash Usage in Signatures

In Bitcoin, signatures are over messages hashed with H_M. Therefore, a second pre-image attack or a collision on H_M can be used to destroy and possibly steal coins: an adversary can ask for a signature on an innocuous transaction (e.g., pay 1 satoshi to address X), but transmit a malicious one instead (e.g., pay 100 BTC to address Y) since there are enough bytes that the adversary controls to guarantee success with high probability.

Signatures of the main hash are also used by Bitcoin developers to transmit alerts. A pre-image attack again does not give useful information to the adversary, as the pre-image always accompanies the signature. Collisions are also not useful, as the adversary cannot sign them. However, a second pre-image allows the adversary to reuse an old signature on a new alert.

4.4 Address Hash

The address hash is used in two contexts. First in the Pay-to-Public-Key-Hash (P2PKH): an address is essentially $y = H_A(p) = \mathrm{RIPEMD160}(\mathrm{SHA256}(p))$ where p is the public key (together with a checksum not used in scripts [3]). Payments to addresses only use the hashed value y, but transactions from an address require the full public key p and the signature on the transaction.

The second use is in Pay-to-Script-Hash (P2SH) scripts. A P2SH is $y = H_A(s)$ where s is a standard script, typically a multi-signature transaction. Payments to a P2SH script do not reveal the pre-image, but transactions spending the coins require it and the signatures of the corresponding parties. Because the only difference between a P2PKH and a P2SH in this context is the number of signatures required, we discuss them jointly here.

Pre-image For previously spent outputs, or for addresses that are being reused, an address hash is already accompanied by its pre-image. A pre-image thus can only reveal the public key(s) for unspent outputs. This has minimal consequences since public keys are not tied to real identities, and hashes are not used for privacy.

Second pre-image An adversary can either attempt to change an existing transaction (in the blockchain, or in the unconfirmed pool), or he can attempt to make his own transactions. In either case, a second pre-image gives the adversary access to a different public key with the same hash. However, because the adversary does not control the corresponding private key of the second pre-image, he cannot use this to his advantage. This is because pre-images are only revealed in combination with signatures.

Collision Collisions are similar, though in this case both public keys are under the adversary's control, and again the adversary does not have access to the private keys. In both scenarios, there is a question of non-repudiation external to the protocol itself: by presenting a second pre-image of a key used to sign a transaction, a user/adversary can claim that his coins were stolen.

Breakage	Effect
Selective forgery	Steal coins from public key
Integrity break	Claim payment not received
Repudiation	-

Table 2: Effects of a break in the signature scheme.

5 Broken Signature Primitives

In this section we describe the use of digital signatures in Bitcoin, and analyze how a break in their unforgeability, integrity, or non-repudiation impacts Bitcoin. We summarize our results in Table 2.

5.1 Digital Signatures in Bitcoin

The digital signature scheme in Bitcoin is the Elliptic Curve Digital Signature Algorithm (ECDSA) with the `secp256k1` [49] parameters and it is used to sign the main hash H_M of transactions. More concretely, a transaction with i inputs and o outputs typically requires a different signature for each of the i inputs. These signatures can be over different parts of the message based on the *hashtype* (see [44]). This can lead to transaction malleability attacks [16], as the same transaction can be encoded multiple ways without invalidating the signature.

The signature scheme is also used for alerts by core developers (with hard-coded public keys) to inform users/clients of critical information, such as a need to upgrade the software due to bugs. The signature is over the main hash H_M of the entire alert structure.

5.2 Modeling Signature Breakage Variants

The security of digital signature schemes is usually discussed in terms of three properties, which can assume different interpretations, but which we define as follows:

1. *Unforgeability* No-one can sign a message m that validates against a public key p without access to the secret key s.
2. *Integrity* A valid signature $\{m\}_s$ does not validate against any $m' \neq m$.
3. *Non-repudiation* A valid signature $\{m\}_s$ does not validate against any public key $p' \neq p$.

where there is an implicit "except with negligible probability", since the messages are hashed.

These properties are linked and a breakage in one usually implies a breakage in the others. In addition, they are often discussed in a much more abstract way: non-repudiation refers to the property that the signature proves to all parties the origin of the signature, but in this case we introduce it in a way that is more akin to Duplicate Signature Key Selection (DSKS) attacks [12].

5.3 Broken Signature Scheme Effects

We now analyze a break in each of these properties separately, starting with the last two, as neither of them can lead to an attack on their own.

Integrity In order for a break in the integrity of the signature scheme to be useful in Bitcoin, a signature of $H_M(m)$ must also be valid for $H_M(m')$. As a result, an adversary must either be able to produce m, m' valid within Bitcoin such that a signature of $H_M(m)$ is also valid for $H_M(m')$ or given $H_M(m)$, the adversary must produce m' with the same effect. Though transactions are already malleable [16], this is not due to ECDSA, but because of the way Bitcoin uses signatures (see [44]). This malleability has been used to cause the issuer of a transaction to think that his payment was not confirmed. Otherwise, both cases involve the main hash in a non-trivial way, so we discuss this case in Section 6.

Non-repudiation For non-repudiation, we consider two types of breakages: either given a message m, one returns two public keys p, p' such that the signature of m under p validates under p' as well, or, given a message m and a public key p, one can find p' to the same end. For regular transactions, even if this is the case, the address hashes of the two public keys must match: $H_A(p) = H_A(p')$ for the scripts to go through, so on its own this is not sufficient. For the alert mechanism, however, if given a message m and a public key p, one can find p' (with its secret key s') such that $\{m\}_{s'}$ validates against p, then an adversary can send fake alert messages. Though these do not alter the behavior in the default client (the "safe mode" that disabled RPC calls is not present after version 0.3.20), they can have an external impact on Bitcoin, e.g., with users manually shutting down clients.

Unforgeability When it comes to unforgeability, we can distinguish between various types of breaks [22]:

1. *Total break* Recover the private key.
2. *Universal forgery* Forge signatures for all messages.
3. *Selective forgery* Forge signature on a message of the adversary's choice.
4. *Existential forgery* Produce a valid signature that is not already known to the adversary.

Because the message to be signed must be the hash of a syntactically valid transaction, an existential forgery is not sufficient, i.e., the probability that it corresponds to a valid message both syntactically and semantically is negligible. Selective forgery on the other hand is sufficient, because it can be used to drain a victim's wallets. From this perspective, the effect of selective forgery and a total break are the same. However, as we discuss later, the type of breakage influences how to upgrade to a new system.

It is worth mentioning that an adversary does not necessarily have access to a user's public key, since addresses that have not been reused are protected by the address

	Signature Property		
Hash Property	**Selective forgery**	**Integrity break**	**Repudiation**
Address Hash (H_A)			
Collision	Repudiate transaction	-	Change existing payment†
Second pre-image	Steal all coins	-	Change existing payment
Pre-image	Steal all coins	-	-
Bounded pre-image	All of the above	-	Change existing payment
Main Hash (H_M)			
Collision	Steal coins	Steal coins†	-
Second pre-image	Steal coins	Double spend†	-
Pre-image	-	-	-
Bounded pre-image	Steal coins	All of the above	-

† Achieving this requires a slight modification of the definitions. See text for details.

Table 3: The effects of a multi-breakage: broken signature scheme in combination with a break in H_A or H_M.

hash H_A. As a result, in the case of a break in the signature scheme, the public key becomes the secret itself, as revealing it can be used to forge signatures!

Finally, note that bad randomness can cause a total break: using the same random number in two different signatures reveals the private key. This weakness was present in some early Bitcoin implementations [48].

6 Multi-Breakage

In this section we analyze how combinations of breakages in different primitives can impact Bitcoin. As shown in the preceding sections, the ways in which a single primitive can break are complementary: having the same primitive break in two different ways gives the attacker the sum of the individual breakages but no more. Moreover, because H_A and H_M are not used together, we only consider a break in the signature algorithm in combination with a break in one of the two hashes. The extra power of our oracle is not needed, so we discuss breakage in terms of the three traditional properties. The results are summarized in Table 3.

6.1 Address Hash and Signature Scheme

Signature Forgery Combining a selective forgery with a *first or second pre-image* break of the address hash can be used to steal all coins that are unspent.

Generating two public keys p, p' with $H_A(p) = H_A(p')$ (*collision*) whose signatures the adversary can forge does not have a direct impact, since the adversary controls both addresses. However, it appears as if two different users are attempting to use the same coin, thus raising a question of repudiation, which we discuss in Section 7.

Signature Integrity As the messages signed for alerts or transactions do not involve H_A, this combination does not increase the adversary's power.

Signature Repudiation A *pre-image* attack on H_A is not useful as the public key is already known.

For a *second pre-image*, assume that given a message m (the hash of a transaction) and a public key p (not under the adversary's control), an oracle returns p' such that $H_A(p) = H_A(p')$ and the signature of m under p also validates against p'. Since the same signature validates for both keys, an adversary can replace p by p' in the unlocking script. Though this does not give the adversary immediate monetary gain, a transaction in the blockchain has been partially replaced.

For *collisions*, assume that given a message m, an oracle returns two public keys p, p' such that $H_A(p) = H_A(p')$ and the signature of m under p validates under p'. If the adversary does not have access to the private keys, he cannot sign the transaction. Otherwise, the effect is identical to the second pre-image case, where the adversary can replace part of a transaction in the blockchain.

6.2 Main Hash and Signature Scheme

Signature Forgery As explained in Section 4.3, none of the potential attacks using the hash H_M required a break in the signature scheme. The partial exceptions were mining under a pre-image break, alerts with collisions, and transactions with second pre-image or collision breaks. We discuss each possibility below.

For *mining*, a pre-image attack is useful when working backwards from a fixed target to get a pre-image for the Merkle root, and turn it into a tree of transactions. The problem identified in Section 4.3 was that there is only negligible probability that the transactions refer to valid, unspent outputs, so a forgery does not solve this issue.

For *alerts*, collisions require forgery. Though the effect of signing and transmitting two different alert messages with the same hash is unclear, it could potentially be used to cause external effects to Bitcoin by making the different messages ask the users to take different actions.

Finally, for *transactions*, collisions and second pre-images on their own can be used to destroy coins, or in appropriate circumstances steal coins. With a forgery, an adversary is guaranteed that he would be able to steal coins no matter what address they went to, as long as it is not protected by the address hash.

Signature Integrity A collision or a second pre-image attack trivially breaks the integrity of the scheme, so we modify the definitions slightly to consider a joint break in the two algorithms.

A *collision integrity* oracle given a public key p produces m, m' such that the signature of $H_M(m)$ is also valid for $H_M(m')$. The adversary can ask for a signature on an innocent transaction, but transmit the malicious one with the still valid signature. Unlike in the regular collision case, the two hashes $H_M(m)$ and $H_M(m')$ are different. Hence, the adversary cannot just replace the transaction in the block, but he can opt never to transmit the innocent one instead.

A *second pre-image integrity* oracle given a public key p and a message m produces m' such that the signature of $H_M(m)$ is also valid for $H_M(m')$. This case also resembles the break on just H_M, but, again, because the hashes are not equal, the adversary cannot simply replace an existing transaction, unless it has not yet been confirmed in a block. This can split the network and destroy coins.

Signature Repudiation The non-repudiation property of the signature scheme necessarily involves a break of H_A, as was explained in Section 5.3. This combination therefore does not increase the adversary's power.

7 Current Bitcoin Implementation

Our analysis in the previous sections reveals properties of the design of the protocol, regardless of which concrete cryptographic building blocks are used. In this section we revisit the current Bitcoin implementation, and in particular its choice of primitives and contingency plans, using the observations from the previous sections.

Breakage	**Effect**
SHA256	
Collisions	Steal coins
Second pre-image	Double spend
Pre-image	Complete failure
Bounded pre-image	All of the above
RIPEMD160	
Any of the above	Repudiate payments
ECDSA	
Selective forgery	Steal coins, Send fake alerts
Integrity break	Claim payment not received
Repudiation	Send fake alerts

Table 4: Effects of concrete primitive breakage on the current version of Bitcoin.

7.1 Current Cryptographic Primitives

In the current implementation of Bitcoin, $H_A(x)$ is instantiated as RIPEMD160(SHA256(x)), and $H_M(x)$ is implemented as SHA256(SHA256(x)). If we translate our design-level analysis from the previous sections into observations on the current primitives, we obtain Table 4.

Specifically, because there are no interesting breaks for H_A, a break in RIPEMD160 is not major cause for concern. Moreover, by our analysis in Section 4.2.1, and because H_M only uses SHA256, an attack against SHA256 is equivalent to an attack against H_M.

7.2 Existing Contingency Plans

A breakage of the cryptographic primitives has interested the community from the early days of Bitcoin. Initial recommendations by Satoshi in informal forum discussions [42] [41] eventually evolved into a "wiki" page which describes contingency plans for "catastrophic failure[s]", including breaks in SHA256 and ECDSA [11]. In this subsection, we present the current high-level plans before we highlight potential pitfalls with the proposed transition mechanisms in Section 7.3. The exact wording of the plans is in Appendix C.

When it comes to the hash functions, the wiki only contains details about a "severe, 0-day failure of SHA-256" [11]. The plan in that case is to switch to a new hashing algorithm, hardcode known public keys with unspent outputs, and hash the old blockchain into a hash tree (under the new hash function), with at least the root being hardcoded into the client.

In the case where "an attacker can sign for a public key that he does not own the private key" [11], the plan depends on whether the attacker can recover the private key and whether there is a stronger algorithm that can use the same key pair. If the attacker cannot recover the private key, and there is a drop-in replacement, the plan is to just switch over to a new algorithm. Otherwise, the new version of Bitcoin "should automatically send old transactions somewhere else using the new algorithm" [11].

There are some commonalities in the two contingency plans. First, both indicate that alerts may be compromised, and point out that notifying users is key. Second, breakage is always defined in terms of an adversary that can defeat the algorithm with "a few days of work". Finally, both plans draw focus on the `OP_CHECKSIG` operation: preventing people from stealing coins is crucial.

7.3 Potential Migration Pitfalls

The contingency plans suggest that "code for all of this should be prepared" [11], but no such mechanism currently exists. Given that a sudden breakage is unlikely, this does not cause immediate concern. However, the size of the codebase suggests that this will not be an easy task, especially since any changes will not be backwards compatible, thus necessitating a hard fork.

Broken RIPEMD160 No plans are in place should RIPEMD160 be broken. As was shown in Section 4.4, this also does not pose a problem, since a break on the address hash does not lead to serious attacks.

Broken SHA256 For a broken SHA256, meaningful collisions or pre-images suggest that new transactions should not be accepted. However, as we saw in Section 4.3, unless a broken hash results in majority power, an adversary cannot alter historical blocks or transactions. The same can be said for hard-coding known public keys with unspent outputs: even if the adversary gets a different key that hashes to the same value, deriving the private key should be infeasible if the signature scheme is still strong. The plans for SHA256 thus seem to be more prudent than necessary, but since they necessitate a hard fork, rehashing the entire blockchain to add new checkpoints or hardcoding public keys can only increase the security of the transition period, but perhaps at a cost of efficiency.

Broken ECDSA For a broken ECDSA, a transition is indeed trivial if the same public key can be kept because the adversary cannot recover the private key. If this is not the case, however, "automatically send[ing] all old transactions somewhere else using the new algorithm" is too vague, since users will need to manually switch over to a new key pair. We will discuss two scenarios for an attacker that can recover private keys: one where the break has already happened, and one where the break has not yet happened, so there is some transitional period.

Imminent ECDSA Break If there is some transition period, developers could force all output scripts to use new-type addresses, so that users could migrate their funds from insecure addresses to secure ones. After a certain period of time, transactions with input scripts using old-style addresses could be rejected, or their security would not be guaranteed. If this period is not long enough, coins may be permanently lost, or stolen if the old-type transactions are still allowed. This is akin to having to a bank retiring old bank notes or a country switching to a new currency (e.g., the Euro), but it shatters the expectation some users might have that their coins will always be accessible, even if they don't access them frequently.

Existing ECDSA Break If the vulnerability is sudden and it reveals the private key, then addresses that have been reused will be entirely unprotected. This is because the public key will have already been revealed in the blockchain, and there is no internal mechanism tying addresses to owners. Any mechanism to protect such addresses would have to be out-of-band, for instance for well-known addresses that can verify their identity (e.g., pools, online wallets, etc.), so that their new keys can be hard-coded into the new client.

Public keys which are protected by the address hash are still safe. In the presence of a trusted central authority, migrating them to a new key would be easy: one needs simply to reveal the key to the trusted authority in order to hardcode it in the new client. However, even if the core developers are trusted, such a solution goes against Bitcoin's decentralized philosophy, and also requires that the public key itself (and by extension the broken private key) is revealed to a third party. It would be interesting to see alternative proposals for this problem, perhaps using commitment schemes or Zero-Knowledge Protocols for hash functions [27].

Overall concerns The migration plans for both the hash and the signature scheme do not clearly distinguish the approaches for a sudden breakage versus a gradual transition, and thus do not fully cover all scenarios. Specifically, in the SHA256 case, they suggest more than is necessary, while in the ECDSA case they do not go far enough: interpreting the plans suggests that either some degree of centralization is necessary, or that some coins will inevitably be lost or stolen.

In general, the plans also do not address when to freeze the blockchain, and whether to roll back transactions in case of a severe 0-day vulnerability. Moreover, if there is a transition period, with old-type transactions using the soon-to-be-broken primitives, and new-type transactions using the stronger primitives, the upgrade plan and the potential for misuse is not clear.

Finally, Bitcoin's defense-in-depth should be improved. For example, the primitives used for hash functions should be combined in a way that prevents single points of failure (see Section 8). In addition, adding a minimum number of transactions in a block and/or a maximum nonce size in coinbase transactions will decrease the probability of an attacker's success and increase the number of calls required to the oracle.

8 Related Work

In spite of the existence of the contingency plans [11] and multiple forum posts pondering the future of Bitcoin should SHA256 break, no other systematic analysis exists. We therefore consider papers which have focused on Bitcoin security in general, and also explore related work focusing on the security of the primitives themselves.

Bitcoin Despite Bitcoin's origins [39] in a non-research environment, Bitcoin's success has sparked multiple papers [13] [50] [20] that have identified or formalized properties such as stability and anonymity in Bitcoin and other cryptocurrencies. Anonymity and privacy issues have also been explored extensively [47] [2] [46] [8].

Research on adversarial miners is also popular. There are infinitely many Nash equilibria for mining strategies [34], and some strategies allow miners controlling $\alpha < 50\%$ of the power to gain disproportionate rewards [19] [15] [18]. Other research has demonstrated that double spending attacks are practical against Bitcoin fast payment scenarios [29] [30], with some further focus on causing a network split [21] or isolating victims from other peers in the P2P nework [25].

[5] focuses on the economics of Bitcoin, including the effect of a history revision, which is discussed in the contingency plans [11]. [16] investigated transaction malleability attacks which were prevalent in 2014.

Cryptographic Primitives For combining hashes, [28] shows that finding simultaneous collisions for multiple hash functions is not much harder than for individual functions. [26] shows that even when the underlying compression functions behave randomly but collisions are easy to generate, finding collisions in the concatenated hash $h_1(x)||h_2(x)$ and the XOR hash $h_1(x) \oplus h_2(x)$ requires $2^{n/2}$ queries. However, when the hash functions use the Merkle-Damgård construction, there is a generic pre-image attack against the XOR hash with complexity $\tilde{O}\left(2^{5n/6}\right)$ [35].

[14] showed that Merkle-Damgård hashes do not behave as random oracles, and neither does $h(h(x))$ [17]. Merkle-Damgård hash functions also behave poorly against pre-image attacks, with the concrete example of finding second pre-images of length 2^{60} for RIPEMD160 in $2^{106} \ll 2^{160}$ time [32]. If an adversary can further find many collisions on a Merkle-Damgård construction, he can also find pre-images that start with a given prefix (Chosen Target Forced Prefix) [31]. This notion was extended in [1] into Chosen Target Forced Midfix attacks and it was proven that at least $2^{2n/3}/L^{1/3}$ queries to the compression function are needed where L is the maximum length of the pre-image.

Attacks against RIPEMD160 and SHA256 only work for a reduced number of rounds. RIPEMD160 pre-images can be computed for 30/80 steps with a complexity of 2^{155} [43], while semi-free-start collisions can be found for 42 steps [37]. For SHA256, collisions exist for 31/64 steps with $2^{65.5}$ operations [36], while pre-images can be found for 45 steps and $2^{255.5}$ time [33].

For the digital signature scheme, [12] showed that certain ECDSA parameters can lead to Duplicate Signature Key Selection, where an adversary can create a different key P' that validates against a correct signature under a key P, which is what we defined as repudiation. [51] described a side-channel attack against the OpenSSL implementation of ECDSA that can recover the private key by snooping on a single signing process, an attack which was also practically demonstrated against Bitcoin [6]. Finally, [7] showed how hash collisions break the security of protocols like TLS, IPSec, and SSH.

9 Conclusions

We have presented the first systematic analysis of the effect of broken primitives on Bitcoin. Our analysis reveals that some breakages cause serious problems for Bitcoin, whereas others seem to be inconsequential.

The main vectors of attack on Bitcoin involve collisions on the main hash or attacking the signature scheme, which directly enable coin stealing. In contrast, a break of the address hash has minimal impact, as addresses do not meaningfully protect the privacy of a user.

Our analysis have also uncovered more subtle attacks. For example, the existence of another public key with the same hash as an address in the blockchain enables parties to claim that they did not make a payment. Such attacks show that the consequences of an attack on a cryptographic primitive can well have social implications rather than technical ones. We leave the economic impact of such attacks as future work.

We have uncovered a worrying lack of defense-in-depth in Bitcoin. In most cases, the failure of a single property in one cryptographic primitive is as bad as multiple failures in several primitives at once. For future versions of Bitcoin, we recommend including various redundancies such as properly combined hash functions.

Bitcoin's migration plans are currently under-specified, and offer at best an incomplete solution if primitives get broken. We offer some initial guidelines for making the cryptocurrency more robust, but there is still a substantial amount of future work to be undertaken.

References

[1] Andreeva, E., and Mennink, B. Provable chosen-target-forced-midfix preimage resistance. In *International Conference on Selected Areas in Cryptography (SAC)* (2011).

[2] Androulaki, E., Karame, G. O., Roeschlin, M., Scherer, T., and Capkun, S. Evaluating user privacy in Bitcoin. In *Financial Cryptography and Data Security (FC)* (2013).

[3] Antonopoulos, A. M. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*, 1st ed. O'Reilly Media, Inc., 2014.

[4] Aurora, V. Lifetimes of cryptographic hash functions. `http://valerieaurora.org/hash.html`, November 17 2012. Accessed: 2016-02-11.

[5] Barber, S., Boyen, X., Shi, E., and Uzun, E. Bitter to better — how to make Bitcoin a better currency. In *Financial Cryptography and Data Security (FC)* (2012).

[6] Benger, N., Pol, J., Smart, N. P., and Yarom, Y. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems (CHES)* (2014).

[7] Bhargavan, K., and Leurent, G. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *Annual Network and Distributed System Security Symposium (NDSS)* (2016).

[8] Biryukov, A., Khovratovich, D., and Pustogarov, I. Deanonymisation of clients in Bitcoin P2P network. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[9] Bitcoin Alert. Some miners generating invalid blocks. `https://bitcoin.org/en/alert/2015-07-04-spv-mining`, July 4 2015. Accessed: 2016-02-11.

[10] Bitcoin Wiki. Protocol rules. `https://en.bitcoin.it/wiki/Protocol_rules`, March 11, 2014. Accessed: 2016-02-11.

[11] Bitcoin Wiki. Contingency plans. `https://en.bitcoin.it/wiki/Contingency_plans`, May 15, 2015. Accessed: 2016-02-11.

[12] Blake-Wilson, S., and Menezes, A. Unknown key-share attacks on the station-to-station (STS) protocol. In *International Workshop on Practice and Theory in Public Key Cryptography (PKC)* (1999).

[13] Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J., and Felten, E. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *IEEE Symposium on Security and Privacy (SP)* (2015).

[14] Coron, J.-S., Dodis, Y., Malinaud, C., and Puniya, P. Merkle-Damgård revisited: How to construct a hash function. In *Annual International Cryptology Conference (CRYPTO)* (2005).

[15] Courtois, N. T., and Bahack, L. On subversive miner strategies and block withholding attack in Bitcoin digital currency. ArXiv e-prints 1402.1718, 2014. `http://arxiv.org/abs/1402.1718`.

[16] Decker, C., and Wattenhofer, R. Bitcoin transaction malleability and MtGox. In *European Symposium on Research in Computer Security (ESORICS)* (2014).

[17] Dodis, Y., Ristenpart, T., Steinberger, J., and Tessaro, S. To hash or not to hash again?(in) differentiability results for H^2 and HMAC. In *Annual International Cryptology Conference (CRYPTO)* (2012).

[18] Eyal, I. The miner's dilemma. In *IEEE Symposium on Security and Privacy (SP)* (2015).

[19] Eyal, I., and Sirer, E. G. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security (FC)* (2014).

[20] Garay, J., Kiayias, A., and Leonardos, N. The Bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)* (2015).

[21] Gervais, A., Ritzdorf, H., Karame, G. O., and Capkun, S. Tampering with the delivery of blocks and transactions in Bitcoin. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[22] Goldwasser, S., Micali, S., and Rivest, R. L. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing (SICOMP)* (1988).

[23] Grover, L. K. A fast quantum mechanical algorithm for database search. In *Annual ACM Symposium on Theory of Computing (STOC)* (1996).

[24] Hearn, M. Why is Bitcoin forking? A tale of differing visions. `https://medium.com/faith-and-future/why-is-bitcoin-forking-d647312d22c1`, August 15, 2015. Accessed: 2016-02-11.

[25] Heilman, E., Kendler, A., Zohar, A., and Goldberg, S. Eclipse attacks on Bitcoin's peer-to-peer network. In *USENIX Security Symposium (USENIX Security)* (2015).

[26] Hoch, J. J., and Shamir, A. On the strength of the concatenated hash combiner when all the hash functions are weak. In *International Colloquium on Automata, Languages and Programming (ICALP)* (2008).

[27] Jawurek, M., Kerschbaum, F., and Orlandi, C. Zero-knowledge using garbled circuits: How to prove non-algebraic statements efficiently. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[28] Joux, A. Multicollisions in iterated hash functions. application to cascaded constructions. In *Annual International Cryptology Conference (CRYPTO)* (2004).

[29] Karame, G. O., Androulaki, E., and Capkun, S. Double-spending fast payments in Bitcoin. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[30] Karame, G. O., Androulaki, E., Roeschlin, M., Gervais, A., and Čapkun, S. Misbehavior in bitcoin: A study of double-spending and accountability. In *ACM Transactions on Information and System Security (TISSEC)* (2015).

[31] Kelsey, J., and Kohno, T. Herding hash functions and the nostradamus attack. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)* (2006).

[32] Kelsey, J., and Schneier, B. Second preimages on n-bit hash functions for much less than 2^n work. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)* (2005).

[33] Khovratovich, D., Rechberger, C., and Savelieva, A. Bicliques for preimages: Attacks on Skein-512 and the SHA-2 family. In *International Workshop on Fast Software Encryption (FSE)* (2012).

[34] Kroll, J. A., Davey, I. C., and Felten, E. W. The economics of Bitcoin mining, or Bitcoin in the presence of adversaries. In *Workshop on the Economics of Information Security (WEIS)* (2013).

[35] Leurent, G., and Wang, L. The sum can be weaker than each part. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)* (2015).

[36] Mendel, F., Nad, T., and Schläffer, M. Improving local collisions: New attacks on reduced SHA-256. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)* (2013).

[37] Mendel, F., Peyrin, T., Schläffer, M., Wang, L., and Wu, S. Improved cryptanalysis of reduced RIPEMD-160. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)* (2013).

[38] Merkle, R. C. A digital signature based on a conventional encryption function. In *Annual International Cryptology Conference (CRYPTO)* (1987).

[39] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, `http://bitcoin.org/bitcoin.pdf`, 2008.

[40] Nakamoto, S. Bitcoin source code v0.1.0: Util.h. `https://github.com/trottier/original-bitcoin/blob/4184ab26345d19e87045ce7d9291e60e7d36e096/src/util.h`, 2009. Accessed: 2016-02-11.

[41] Nakamoto, S. Dealing with SHA-256 collisions (msg #6). `https://bitcointalk.org/index.php?topic=191.msg1585#msg1585`, June 14 2010. Accessed: 2016-02-11.

[42] Nakamoto, S. Hash() function not secure (msg #28). `https://bitcointalk.org/index.php?topic=360.msg3520#msg3520`, July 16 2010. Accessed: 2016-02-11.

[43] Ohtahara, C., Sasaki, Y., and Shimoyama, T. Preimage attacks on step-reduced RIPEMD-128 and RIPEMD-160. In *International Conference on Information Security and Cryptology (Inscrypt)* (2010).

[44] Okupski, K. Bitcoin developer reference working paper. `http://enetium.com/resources/Bitcoin.pdf`, 2015. Accessed: 2016-02-11.

[45] Proos, J., and Zalka, C. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Information & Computation* (2003).

[46] Reid, F., and Harrigan, M. An analysis of anonymity in the Bitcoin system. In *Security and Privacy in Social Networks* (2013).

[47] Ron, D., and Shamir, A. Quantitative analysis of the full Bitcoin transaction graph. In *Financial Cryptography and Data Security (FC)* (2013).

[48] Schneider, N. Recovering Bitcoin private keys using weak signatures from the blockchain. `http://www.nilsschneider.net/2013/01/28/recovering-bitcoin-private-keys.html`, January 28 2013. Accessed: 2016-02-11.

[49] Standards for Efficient Cryptography. Sec 2: Recommended elliptic curve domain parameters version 2.0. `http://www.secg.org/sec2-v2.pdf`, 2010.

[50] Tschorsch, F., and Scheuermann, B. Bitcoin and beyond: A technical survey on decentralized digital currencies. Cryptology ePrint Archive, Report 2015/464, 2015. `https://eprint.iacr.org/2015/464`.

[51] Yarom, Y., and Benger, N. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. `https://eprint.iacr.org/2014/140`.

A Transaction Verification

Bellow is the full procedure for verifying a transaction for inclusion in the blockchain as of January 2016 [10].

1. Check syntactic correctness
2. Make sure neither in or out lists are empty
3. Size in bytes < MAX_BLOCK_SIZE
4. Each output value, as well as the total, must be in legal money range
5. Make sure none of the inputs have hash=0, n=-1 (coinbase transactions)
6. Check that nLockTime <= INT_MAX, size in bytes >= 100, and sig opcount <= 2
7. Reject "nonstandard" transactions: scriptSig doing anything other than pushing numbers on the stack, or scriptPubkey not matching the two usual forms
8. Reject if we already have matching tx in the pool, or in a block in the main branch
9. For each input, if the referenced output exists in any other tx in the pool, reject this transaction.
10. For each input, look in the main branch and the transaction pool to find the referenced output transaction. If the output transaction is missing for any input, this will be an orphan transaction. Add to the orphan transactions, if a matching transaction is not in there already.
11. For each input, if the referenced output transaction is coinbase (i.e., only 1 input, with hash=0, n=-1), it must have at least COINBASE_MATURITY (100) confirmations; else reject this transaction
12. For each input, if the referenced output does not exist (e.g., never existed or has already been spent), reject this transaction
13. Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in legal money range
14. Reject if the sum of input values < sum of output values
15. Reject if transaction fee (defined as sum of input values minus sum of output values) would be too low to get into an empty block
16. Verify the scriptPubKey accepts for each input; reject if any are bad
17. Add to transaction pool
18. "Add to wallet if mine"
19. Relay transaction to peers
20. For each orphan transaction that uses this one as one of its inputs, run all these steps (including this one) recursively on that orphan

B Block Verification

Bellow is the full procedure for verifying a block in the blockchain as of January 2016 [10]. The procedure is split into logical chunks. Verification always starts with the main procedure.

B.1 Main procedure

1. Check syntactic correctness
2. Reject if duplicate of block we have in any of the three categories
3. Transaction list must be non-empty
4. Block hash must satisfy claimed nBits proof of work
5. Block timestamp must not be more than two hours in the future
6. First transaction must be coinbase (i.e., only 1 input, with hash=0, n=-1), the rest must not be
7. For each transaction, apply "tx" checks 2-4
8. For the coinbase (first) transaction, scriptSig length must be 2-100
9. Reject if sum of transaction sig opcounts > MAX_BLOCK_SIGOPS
10. Verify Merkle hash
11. Check if prev block (matching prev hash) is in main branch or side branches. If not, add this to orphan blocks, then query peer we got this from for 1st missing orphan block in prev chain; done with block
12. Check that nBits value matches the difficulty rules
13. Reject if timestamp is the median time of the last 11 blocks or before
14. For certain old blocks (i.e., on initial block download) check that hash matches known values
15. Add block into the tree. There are three cases:
    (a) block further extends the main branch; (see Appendix B.2)
    (b) block extends a side branch but does not add enough difficulty to make it become the new main branch; (see Appendix B.3)
    (c) block extends a side branch and makes it the new main branch. (see Appendix B.4)
16. For each orphan block for which this block is its prev, run all these steps (including this one) recursively on that orphan

B.2 Case 1: Extend main branch

For case 1, adding to main branch:

1. For all but the coinbase transaction, apply the following:
    (a) For each input, look in the main branch to find the referenced output transaction. Reject if the output transaction is missing for any input.
    (b) For each input, if we are using the nth output of the earlier transaction, but it has fewer than n+1 outputs, reject.
    (c) For each input, if the referenced output transaction is coinbase (i.e., only 1 input, with hash=0, n=-1), it must have at least COINBASE_MATURITY (100) confirmations; else reject.
    (d) Verify crypto signatures for each input; reject if any are bad
    (e) For each input, if the referenced output has already been spent by a transaction in the main branch, reject
    (f) Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in legal money range
    (g) Reject if the sum of input values < sum of output values
2. Reject if coinbase value > sum of block creation fee and transaction fees
3. (If we have not rejected):
4. For each transaction, "Add to wallet if mine"
5. For each transaction in the block, delete any matching transaction from the transaction pool
6. Relay block to our peers
7. If we rejected, the block is not counted as part of the main branch

B.3 Case 2: Extend short side branch

For case 2, adding to a side branch, we don't do anything.

B.4 Case 2: Extend long side branch

For case 3, a side branch becoming the main branch:

1. Find the fork block on the main branch which this side branch forks off of
2. Redefine the main branch to only go up to this fork block
3. For each block on the side branch, from the child of the fork block to the leaf, add to the main branch:
    (a) Do "branch" checks 3-11
    (b) For all but the coinbase transaction, apply the following:
        i. For each input, look in the main branch to find the referenced output transaction. Reject if the output transaction is missing for any input.
        ii. For each input, if we are using the nth output of the earlier transaction, but it has fewer than n+1 outputs, reject.
        iii. For each input, if the referenced output transaction is coinbase (i.e., only 1 input, with hash=0, n=-1), it must have at least COINBASE_MATURITY (100) confirmations; else reject.
        iv. Verify crypto signatures for each input; reject if any are bad
        v. For each input, if the referenced output has already been spent by a transaction in the main branch, reject

        vi. Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in legal money range
        vii. Reject if the sum of input values < sum of output values
    (c) Reject if coinbase value > sum of block creation fee and transaction fees
    (d) (If we have not rejected):
    (e) For each transaction, “Add to wallet if mine”

4. If we reject at any point, leave the main branch as what it was originally, done with block
5. For each block in the old main branch, from the leaf down to the child of the fork block:
    (a) For each non-coinbase transaction in the block:
        i. Apply “tx” checks 2-9, except in step 8, only look in the transaction pool for duplicates, not the main branch
        ii. Add to transaction pool if accepted, else go on to next transaction
6. For each block in the new main branch, from the child of the fork node to the leaf:
    (a) For each transaction in the block, delete any matching transaction from the transaction pool
7. Relay block to our peers

C Contingency Plans

Below are the full contingency plans verbatim from the Bitcoin Wiki [11] as of January 2016 should the two main primitives be broken:

C.1 SHA-256 is broken

Situation

Severe, 0-day failure of SHA-256. First/second preimage resistance or collision resistance can be defeated with only a few days of work.

Impact

- Attacker may be able to defeat OP_CHECKSIG, which hashes transactions before signing.
- Attacker may be able to split the network by creating identical transactions or blocks with the same hashes.
- Attacker may be able to create blocks very quickly.
- The alert system may be compromised.

Response

- Users will be notified to shut down their clients. Note that the attacker may be able to send valid alerts, which could disrupt notification efforts.
- OP_CHECKSIG will be changed to use some other hash outside of old blocks.
- All addresses in the version-1 chain that have a known public key and at least one unspent output will have their public keys hardcoded into the client. When a version-2 transaction spends one of these version-1 outputs, the hardcoded public key will be used instead of the hash.
- The version-1 chain will be securely hashed into a hash tree. At least the root of the version-1 tree will be hardcoded into the client.
- All hashing Bitcoin does will use the new hashing algorithm.

[Code for all of this should be prepared.]

C.2 ECDSA is broken

Situation

An attacker can sign for a public key that he does not own the private key for in only a few days of work.

Impact

- Attacker can spend money that is not his in a large number of cases. Transactions to addresses that have never been used before may be protected if SHA-256 and RIPEMD-160 are still strong.
- Alert system may be compromised.

Response

If the attacker can’t get the private key from the public key easily and a stronger algorithm that can use ECDSA keys is available:

- Switch to the stronger algorithm.
- Get users to update. Alerts will be compromised.

Otherwise:

- OP_CHECKSIG should use some other signing algorithm.
- As soon as the new version of Bitcoin is run, it should automatically send all old transactions somewhere else using the new algorithm.
- Get users to update immediately. Alerts will be compromised.

[Code for all of this should be prepared.]