LEDGER
ledgerjournal.org

ISSN 2379-5980 (online)
DOI 10.5915/LEDGER.2018.101

RESEARCH ARTICLE

Data Insertion in Bitcoin's Blockchain

Andrew Sward,† Ivy Vecna,‡ Forrest Stonedahl§*

Abstract. This paper provides the first comprehensive survey of methods for inserting arbitrary data into Bitcoin's blockchain. Historical methods of data insertion are described, along with lesser-known techniques that are optimized for efficiency. Insertion methods are compared on the basis of efficiency, cost, convenience of data reconstruction, permanence, and potentially negative impact on the Bitcoin ecosystem.

1. Introduction

From its genesis block, and the now infamous headline that Satoshi chose to inscribe as the first permanent message in the Blockchain, Bitcoin has been utilized as a free speech platform.[1, 2] In addition to exchanging digital currency on a global scale, Bitcoin also provides users with the ability to publish information that cannot be censored or retracted, and will be *permanently* available to the world (as long as Bitcoin itself persists).[3] However, the Bitcoin community is divided with regard to whether this use of Bitcoin as a platform for data publication/storage is an appropriate one:

> "The use of bitcoin's blockchain to store data unrelated to bitcoin payments is a controversial subject. Many developers consider such use abusive and want to discourage it. Others view it as a demonstration of the powerful capabilities of blockchain technology and want to encourage such experimentation."
>
> -Andreas Antonopoulos[4]

Everyone has their own vision of what Bitcoin can and should be used for. While we are inclined to favor the view that the insertion of data *can be* a legitimate and valuable use of the Blockchain, the purpose of this article is not to argue in favor of (or against) the practice; rather, it is to enumerate the historical and efficient methods of data publication, and to examine the benefits and drawbacks corresponding to each method. Specifically, we will compare data publication methods on the basis of efficiency, cost, convenience of data reconstruction, permanence, and the potentially negative impact on the Bitcoin ecosystem.

We believe this work will be of interest to several audiences:

† A. P. Sward (andrewsward@augustana.edu) is Assistant Professor of Applied Mathematics at Augustana College, IL
‡ I. Vecna (ivyvecna15@augustana.edu) is an undergraduate researcher at Augustana College, IL
§ F. Stonedahl (forreststonedahl@augustana.edu) is Assistant Professor of Computer Science at Augustana College, IL
* Augustana College Cryptocurrency Analytics Lab: 1Data2DYNE9ajurrnDmqZ8xNU1GzXsa9E9

(1) For those who wish to store data in the Blockchain, we identify which methods optimize data storage (and minimize the associated cost) given the constraints of the protocol.
(2) For those who are concerned that the Blockchain is being “co-opted” for data publication/storage, we provide a clear outline of the presently-available methods and explain which methods mitigate negative side effects for other users.
(3) For future digital archeologists, we provide a valuable point of reference that may allow posterity to unearth virtual artifacts that otherwise might remain hidden forever in the Blockchain in binary format.[5]

2. Related Work

It is common knowledge that extrinsic data can be stored in the Blockchain, and there are numerous websites that provide access to a subset of that data,[6,7,8] and some excellent sleuthing has uncovered a variety of interesting historical artifacts that have previously been stored.[9] Nevertheless, there remains confusion and misinformation about the variety of different methods by which data can be (and has been) stored. For instance, a recent comprehensive textbook on Bitcoin included the following:

> “There’s no good way to prevent people from writing arbitrary data into the Bitcoin block chain [sic]. One possible countermeasure is to only accept Pay-to-Script-Hash transactions. This would make it a bit more expensive to write in arbitrary data, but it still wouldn’t prevent it.”[10]

The first claim, that one cannot prevent arbitrary data insertion, is correct, since there is no general way to distinguish between legitimate address hashes and arbitrary binary data. However, the second claim is false, as P2SH (Pay-to-Script-Hash) transactions actually provide the *least expensive* and most efficient methods for storing large amounts of arbitrary data (see section 5).

There are also a variety of websites that provide user-friendly tools to publish data of one’s choice.[11,12] However, these tools are currently using the Pay-to-Fake-Key-Hash (P2FKH) method, which has serious drawbacks (discussed in section 4.2) that make it inefficient for users of the service and harmful to the Bitcoin infrastructure.

While some previous works have analyzed the graph structure and anonymity of Bitcoin’s transaction ledger,[13, 14] there is a dearth of academic work studying the publication and storage of arbitrary data, with only a few notable exceptions. In 2015, apparently unaware that Bitcoin users were already embedding arbitrary (ASCII and binary) data into Blockchain transactions, Sleiman et al.naively proposed a protocol and developed software for including text messages in the Blockchain by using the transaction currency *amount* field to encode data.[15] The result was a highly inefficient publication mechanism that could only store up to 8 lowercase English letters per transaction output. More recently, Bartoletti and Pompianu analyzed the metadata attached to transactions that use one specific data insertion method (`OP_RETURN`, see section 4.5) to build protocol layers on top of the Bitcoin protocol (*e.g.*, for asset management, notarization, etc...).[16] In a different vein, Permacoin proposed the idea of building an alternative to Bitcoin that uses “proof-of-retrievability” rather than proof-of-work, which would (by design) allow for the storage of massive amounts of arbitrary data, extrinsic to the transaction ledger.[17]

3. Background: The Bitcoin Script Language

3.1. Standard Scripts—Bitcoin's stack-based scripting language for creating transactions is simply called "Script." Bitcoin transactions contain input scripts and output scripts. The input scripts are solutions (unlocking scripts) to previous output scripts (locking scripts) in prior transactions stored in the Blockchain.[18] There are currently 5 *standard* script types that are used and accepted on the Bitcoin network for transactions.[19, 20] The standard script types include Pay-to-Public-Key (P2PK), Pay-to-Public-Key-Hash (P2PKH), Multi-Signature, Pay-to-Script-Hash (P2SH), and `OP_RETURN` (see Appendix B for the Script formats). Sections 4 and 5 will demonstrate how each of these script types can be used to store arbitrary data in Bitcoin's blockchain.

3.2. Methods—For analysis of historical methods and testing each of the data insertion methods in this paper, we used the open-source Java library, BitcoinJ.[21]With this tool we iterated through the Blockchain and searched for scripts that do not fit the standard script types, as well as specific script formats. We also used BitcoinJ to build scripts to use in our own transactions, and we tested these by broadcasting them to the Bitcoin network (through Blockchain.info).[22] Example code for iterating through the Blockchain and building scripts can be found in Appendix D.

3.3. Technical Limitations on Scripts and Transactions—At the time of writing, a *standard* Bitcoin transaction is limited to 100 KB, each input script is limited to 1650 bytes,[23] and any single element being pushed onto the execution stack is limited to 520 bytes. After script execution, the stack must contain exactly one non-false element.[24] Input scripts may not contain any OP codes other than `OP_PUSHDATA` (except within the special *Redeem Script* portion of a P2SH). The minimum output value (*min non-dust* – see definition in Appendix A) for a P2PKH is currently 546 satoshis. Transactions that deviate from these rules are considered non-standard and will not be picked up by most miners.[25]

3.4. Standard Script Enforcement—Most of the above restrictions on scripts are enforced by a method in the Bitcoin Core source code called isStandard().[26] These limitations were imposed in the Bitcoin Core client for a variety of reasons, including performance considerations and preventing an issue known as *transaction malleability* (see section 6.2). However, this severely restricts the input and output scripts that one can write. An input script that spends a P2SH transaction is the only place that affords some flexibility in the use of the Bitcoin Script language. This flexibility allows more complex logical operations for financial transactions, and it also allows the greatest variety of data insertion mechanisms. We will first explain each of the four simpler non-P2SH methods (section 4) then explain the more sophisticated P2SH-based methods (section 5).

4. Data Insertion Methods Not Involving P2SH

4.1. Coinbase—The *coinbase data* is the content of the input of a generation transaction. The coinbase data is arbitrary and can be up to 100 bytes in size.[27, 28] The coinbase data has been left to the discretion of the miners and has typically been a field where miners insert ASCII encoded strings declaring the name of their mining pool, or other short messages. The coinbase data is also used by miners to signal support for various proposed changes to the Bitcoin protocol. Some, if not all, of the coinbase data may be commandeered by developers in future versions of the Bitcoin protocol. While this field *is* a way of storing arbitrary data in the Blockchain, it is available only to miners and not general Bitcoin users; it is therefore included in this paper for thoroughness, but will not be mentioned again.

4.2. P2FKH—A very common and controversial data insertion method utilizes the standard Pay-to-Public-Key-Hash script, storing the data in the `<PubKeyHash>` field of the output script along with a non-dust amount of Bitcoin to "burn." We refer to this as Pay-to-Fake-Key-Hash (P2FKH). The user does not have a public key that would hash to the data they are storing; because of this, these transaction outputs can never be spent. However, because they are valid Unspent Transaction Outputs (UTXOs – see Appendix A) and the miners have no way of knowing whether the hash corresponds to a real public key that someone possesses, the miners must keep track of these UTXOs (forever). The storage afforded by the P2FKH method is 20 bytes per output, but many outputs can be included in a single transaction. This method has been used to store text,[29] images (see Fig. 1), and mp3 files in Bitcoin's blockchain[30] and is currently the method employed by tools like Apertus.io.[11]

Fig. 1. This JPEG image of Nelson Mandela was stored on 7 December 2013 as P2FKHs spread across multiple transactions, within block 273,536. Size: 14,400 bytes.[31]

4.3. P2FK—Data can also be stored as a fake public key (P2FK), instead of a fake public key hash. An uncompressed public key is 65 bytes,[32] and the overall script has 3 fewer OP codes, making this a much more efficient method for data storage than P2FKH. However, it does not seem to be in prevalent use by the community as a method for storing data. One possible reason for this is that it would be relatively easy for nodes to detect fake (uncompressed) public keys and the Bitcoin developers (or miners) could shut down this approach in the future.[33] Storing data using a fake compressed public key (33 bytes) could work around this, and would still provide more data efficiency than P2FKH. However, this method also suffers from the problem of creating unspendable UTXOs.

4.4. Concerns caused by fake addresses—Both the popular P2FKH and the P2FK methods are problematic for several reasons:

(1) Their storage methods are inefficient, incurring greater overhead (particularly P2FKH) and more UTXOs than necessary.
(2) The miners must permanently track each unspendable UTXO created this way.

(3) These methods irretrievably "burn" Bitcoin. P2FKH and P2FK both require the user to send a small amount of Bitcoin (greater than or equal to the min non-dust value) to each fake address.
(4) Storing arbitrary data in the Blockchain will create "bloat" to the overall ledger size.

The first three problems can be addressed by using improved data storage methods. The fourth objection will apply to any data insertion method, and the Blockchain is destined to grow larger as long as blocks are mined and transactions are occurring, regardless of what the transactions themselves actually represent. Whether the value of the data being stored is a worthwhile use of the Bitcoin network's resources is a point the community will continue to debate. Regardless of the data storage use case, Bitcoin will face scalability issues, which developers are already attempting to address (*e.g.*, segwit,[34] Peter Todd's Merkle Mountain Range proposal to use commitments to obviate the need to store the full UTXO set [35]).

Table 1. BTC (Ƀ) burned storing ASCII in P2FKH transactions

ASCII Character Threshold	Amount Burned Via P2FKH	Amount of ASCII Data Stored	UTXOs
18	118.96Ƀ	2.59 MB	129,410
19	62.77Ƀ	2.58 MB	129,004
20	62.54Ƀ	2.57 MB	128,521

Table 1 shows an estimated amount of Bitcoin that has been burned to fake mostly-text addresses using the P2FKH method, as of 7 June 2017. Specifically, we aggregated the balances for all P2PKH UTXOs for which the address has never been used as an input script, and the key hash contains 18 (or more) consecutive bytes from the set of printable ASCII characters, plus tabs, newlines, and null ('\x00') characters that may have been used as padding around textual data.[36]

4.5. OP_RETURN—The `OP_RETURN` standard script was added as a response to the increasing numbers of users using P2FKH to store data (or metadata) in transactions.[37]`OP_RETURN` allows a small amount of data to be included in each transaction, creating a *provably unspendable* UTXO that the miners do not need to track, and that does not require a non-dust burn value.

There can be many outputs in a single Bitcoin transaction, but only one of these can be an `OP_RETURN` in a standard transaction.[38] `OP_RETURN` can currently only store 80 bytes per transaction. This limit has fluctuated over time (see Bartoletti and Pompianu for a discussion about the history of `OP_RETURN`[16]). To use more than one `OP_RETURN` multiple transactions are required.[39] The order in which these transactions are mined by the decentralized Bitcoin network is difficult to control. Overall this method is appropriate for inserting small amounts of data (or transaction metadata), but it is not suitable for large quantities of data. Some community members have also expressed concern about the robustness of storing data using `OP_RETURN`, since provably unspendable UTXOs can be pruned by nodes, and may not be permanently stored/distributed by as many nodes.[40]

4.6. P2FMS—Another data insertion method (Pay-to-Fake-Multisig) that commonly appears in the Blockchain is a 1-of-2 or 1-of-3 multisig script,[41] with one real public key, and 1 or 2 fake keys containing arbitrary data.[42] Because these transactions are spendable, a user can avoid creating UTXO bloat. For the lowest overhead cost, one would use a (real) compressed public key, and store the data using two fake uncompressed public keys (65 bytes each). This method would keep the data in the UTXO set only until the user decides to spend these outputs (using the one real key). Multiple P2FMS outputs can be stored within a single transaction, consistently using the same real public key in all of them, making data reconstruction straightforward.

However, transactions containing a single `OP_CHECKMULTISIG` must be larger than 400 bytes; specifically, the default requirement is 20 bytes per sigop,[43] and one instance of `OP_CHECKMULTISIG` counts as 20 sigops.[44] This limitation makes redemption of these UTXOs uneconomical: the cost in fees for spending these UTXOs will be greater than the min non-dust values that would typically be sent to them.[45] Therefore, users with no regard for the UTXO bloat can simply use all 3 pubkey fields to store arbitrary data with a burn amount.

5. Data Insertion Methods Using P2SH

5.1. P2FSH—Similar to P2FKH, the Pay-to-Fake-Script-Hash (P2FSH) method simply stores data as a fake hash. P2FSH requires two fewer OP codes than P2FKH (making it slightly more efficient) but still creates an unspendable UTXO. The remainder of section 5 is dedicated to methods that store data in the input script that spends a P2SH output, rather than in the output script.

5.2. Two Stages of P2SH Transactions—There are two stages of P2SH: creating the UTXO and spending the UTXO. To create a P2SH UTXO, the user first creates a Redeem Script, and then applies the HASH160 algorithm to this script.[46] The output script is then:

```
OP_HASH160 <RedeemScriptHash> OP_EQUAL
```

To spend this UTXO, the user creates an input script (referencing the UTXO above) consisting of the Redeem Script itself (as a single stack element, thus limited to 520 bytes) preceded by a sequence of Script operations that will make the Redeem Script result in only true after execution.[47] There are two approaches to data insertion: either store arbitrary data *inside* the Redeem Script itself, and/or store arbitrary data in the portion of the input script that precedes the Redeem Script. For instance, a user might simply make a Redeem Script that contains an `OP_PUSHDATA2` (3 bytes) followed by a 517-byte data element.[48] Since any stack element other than `OP_0` is evaluated as "true," this script will successfully redeem the UTXO. However, because of the 520-byte Redeem Script limit, it is more efficient to store large amounts of data in the portion of the input script that precedes the Redeem Script (see Fig. 6 for a visual representation). We will next discuss such methods (see Appendix C for the full scripts). Variations of the following P2SH-based methods have been used to store data in the Blockchain since June 2014.[49]

5.3. Data Drop Method—The Data Drop method pushes data onto the stack and drops it off the stack during script execution, typically with the use of the OP DROP operation. Consider the following Redeem Script: `OP_DROP ... OP_DROP <PubKey> OP_CHECKSIG`.[50] The preceding input script operations are then `<Sig> <Data>...<Data>`. The stored data must be split into chunks of at most size 520 bytes each. The signature is 71-73 bytes and the Redeem Script is 37 bytes, which leaves 1529 bytes for arbitrary data after accounting for the pushdata OP codes. Recall the input script is constrained by the input size limit of 1650 bytes (see section 3.3), but these inputs can be chained together within a single transaction (up to the 100 KB TX size limit) to store large amounts of data in a nearly contiguous and easy-to-reconstruct format (more about reconstruction in section 8). This method has been used to store relatively large image files within a single transaction in the Blockchain (see Fig. 2).

We include a compressed `<PubKey>` as part of the Redeem Script to ensure that the Redeem Script hashes to something new each time this method is used with a new key, and the use of a signature (`<Sig> ... OP_CHECKSIG`) prevents a double-spend attack (see section 6.1). The data insertion method that provides the lowest known overhead (and publication cost) is a variant of this (Data Drop w/o Sig) that eschews the use of signatures and keys in order to pack more data into each transaction input, at the cost of potential adversarial tampering. *However,* even using signatures, an adversary could perform an online attack to tamper with data stored using the Data Drop method (see section 6.2). The trade-off between maximizing storage capacity and ensuring transaction security and data integrity is discussed further below.

Fig. 2. JPEG image of Mr. Burns stored (without any burns) in a single transaction on 5 April 2017 using the Data Drop w/ Sig method (multiple P2SH inputs with a Redeem Script of OP_DROPs). Size: 34,600 bytes.[51]

5.4. Data Hash Method—The Data Hash method is a more sophisticated method for inserting data in the Blockchain.[52] The largest input script in Blockchain history is an example of this script type; this transaction was included on 27 November 2014, by an unknown author.[53, 54] This transaction included a parody of a Western Union advertisement (see Fig. 3). Similar to the Data Drop method, the input script preceding the Redeem Script contains repeated chunks of `<Data>...<Data>`. The Redeem Script is of the form:

```
OP_HASH160 <DataElementHash> OP_EQUALVERIFY
```

These three commands are then repeated for each data element that is pushed onto the stack by the input script. Rather than merely dropping each data element off the stack, this script uses hashes to verify that each chunk of data has not been tampered with. Since the hashes are stored in the Redeem Script, and the hash of the Redeem Script was recorded in the first stage UTXO, no other data can be substituted into the input script that spends this UTXO, even if the inputs for this transaction were not signed. However, signing each input (by inserting

`<Sig>` at the beginning of the input script and `<PubKey> OP_CHECKSIG` at the end of the Redeem Script) is still necessary to prevent an adversary from potentially reordering the inputs, or including a subset of the inputs, in a competing transaction. These security concerns are further discussed in the next section.

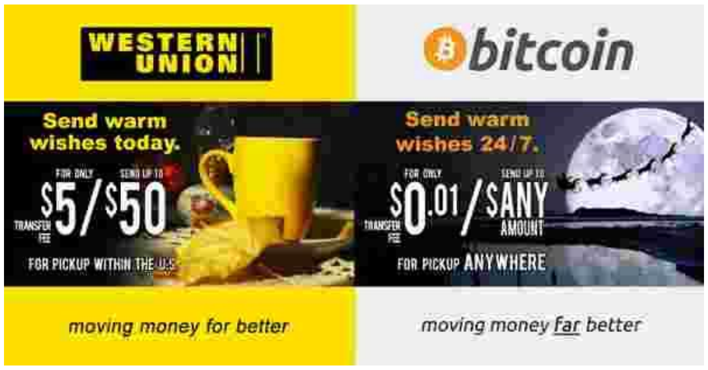

Fig. 3. This JPEG image is stored in Bitcoin's blockchain as a GZIP archive file inside *one* input script of a P2SH output. (Compressed) size: 9,265 bytes. This input script is the largest input script present in the Blockchain to date.[55]

6. Security and Data Integrity

6.1. Sniping UTXOs—We refer to *sniping* as the process of re-appropriating a transaction's unsigned inputs to a new transaction with different outputs (created by the sniper and broadcast simultaneously) to hijack the funds those inputs represent.[56] Only one of these double-spend attempts may be included in the Blockchain. Signatures are designed to protect against sniping because they prohibit adversaries from making any changes to the signed portion of the transaction (to do so would require generating a new valid signature, which the adversary cannot do without the user's private key). However, when a user creates a signature for an input script, the output scripts are secured, but not the input scripts.[57]

Redeem Scripts that do not require a signature are thus vulnerable. If such a script is used multiple times, it may become associated with its hash, and UTXOs that use this hash may be spent by anyone who provides the corresponding Redeem Script. One could include a unique element in the Redeem Script so that the hash (of the Redeem Script) is different with each use.[58] These transactions, however, could still be sniped in real-time by sophisticated bots.

6.2. Transaction Malleability—We define transaction malleability to mean *any* change to a transaction that is broadcast (prior to block acceptance). Transaction malleability is a problem that has plagued Bitcoin for years, and has been addressed in a variety of ways by the Bitcoin Core development team. The threat to normal users is now rarely more than an annoyance but for data publishers, it is a potentially severe problem that warrants discussion.[59]

When a new transaction is broadcast to the P2P Bitcoin network, it gets passed from node to node, with nodes verifying it and storing it into the mempool of possible transactions to include in a block. An adversarial node may receive a transaction and create a modified version of this transaction to pass along to others in the network. These changes may be as innocuous as changing a PUSHDATA OP code,[60] but a more detrimental change could be to alter the arbitrary data stored using the Data Drop method. As long as the scripts themselves still result in valid execution, the modified transaction will have a new transaction ID and could be included in the Blockchain in this modified form. No "functional" transaction data has been changed: the inputs and outputs are still accounted for correctly.

Since the Data Drop with signatures method prevents sniping, it does not currently appear to be a target for malicious agents.[61] However, the DataDrop method includes no measures to prevent an agent on the network from modifying the arbitrary data a user is trying to store, even if each input is signed. In contrast, the Data Hash method ensures data integrity because the hash of each data element is checked during execution of the Redeem Script. While a Data Hash transaction that does not contain a signature could be easily sniped, the sniper would still have to include the exact unmodified data as input. However for data spanning multiple (unsigned) inputs, a sniper could *rearrange* the inputs, or only spend some of the inputs and not others, causing the data to be stored in the Blockchain in an unintended order. Adversaries motivated by mere financial gain can be discouraged by assigning only the min non-dust Bitcoin value for each (unsigned) P2SH input, making the sniper effectively pay more in fees (to store your desired data) than they would recoup from redirecting the output to their own address.[62] Thus, the only method guaranteed to preserve data integrity when using multiple outputs is Data Hash *with signatures*.

None of the simpler data insertion methods (P2FKH, P2FK, P2FMS, P2FSH, OP_RETURN) suffer from malleability or sniping concerns, since the data is stored within signed outputs.[63]

Table 2. P2SH-based Data Insertion Method Summary (Single Input)

Method	SigScript*	RedeemScript*	Max Data	Integrity	Snipeable
Data Drop (w/o sig)	`<Data>`	`OP_DROP...`	1630	No	Yes
Data Drop (w/ sig)	`<Sig><Data>`	`OP_DROP... OP_CHECKSIG`	1529	No	No
Data Hash (w/o sig)	`<Data>`	`OP_HASH <DataHash> OP_EQV...`	1560	Yes**	Yes**
Data Hash (w/ sig)	`<Sig><Data>`	`OP_HASH <DataHash> OP_EQV... OP_CHECKSIG`	1461	Yes	No

* See Appendix C for the full scripts used for these calculations.

** If sniped, multiple inputs within the transaction can be reordered, even though the data within each input cannot be changed.

Table 2 summarizes the two P2SH-based methods with and without signatures in terms of security and data capacity. Although the Data Hash w/ Sig method provides the least data capacity of these methods, the benefit of guaranteed data integrity likely outweighs the loss of efficiency.

7. Efficiency Comparison and Costs

First, regarding efficiency concerns about bloating the UTXO set using fake addresses in UTXOs (as discussed in section 4.4), which impacts the scalability of the Bitcoin ecosystem:

- P2FKH and P2FSH are both extremely wasteful, providing only 20 bytes of data per unspendable UTXO.
- P2FK is also quite wasteful, although using uncompressed keys currently affords 65 bytes of data per unspendable UTXO.[64]
- The currently allowed form of P2FMS (with all 3 addresses fake) could store as much as 195 bytes (using 3 uncompressed keys) per unspendable UTXO. Versions of P2FMS with 1 real key are spendable, but there is currently no economic benefit to retrieve min non-dust values.
- OP_RETURN does not bloat the UTXO set, since it is provably unspendable and nodes may prune it.
- Both forms of the P2SH-based methods that store the data in input scripts (Data Drop and Data Hash) do not increase the UTXO set at all, since all created TXOs get redeemed.

Next, we consider two additional measures of efficiency:

(1) The total amount of data (*i.e.* including overhead) that is required to be added to the Blockchain in order to store a specified amount of arbitrary data (shown in Fig. 4 and Table 3). This relates to scalability issues, and will be of interest to those concerned with storing full copies of the Blockchain.

(2) The total cost in satoshis, using current minimal (20 satoshis/byte) fee and min non-dust burn rates necessary for a transaction to be accepted, for storing a specified amount of arbitrary data (shown in Fig. 5 and Table 3).[65] This measure is of interest to those who wish to store data in the Blockchain inexpensively.

Table 3. Method Summary (Max Size and Cost) for a Fee of 20 Satoshi/byte

Method	Stored in UTXO Set?	Max Data Per TX*	Total Cost **	Data Eff.	Cost Eff.***
PF2KH	Yes	58,680	.03601624	58.7%	61.30
PF2K	Yes	85,280	.02715132	85.3%	31.80
OP_RETURN	Prunable	80	.00006340	25.2%	79.25
P2FMS	Yes	92,624	.02522220	92.6%	27.23
P2FSH	Yes	62,340	.03701302	62.3%	59.37
Data drop (w/o Sig)	No	96,060	.02042260	94.1%	21.26
Data Drop (w/ Sig)	No	90,099	.02042260	88.2%	22.66
Data Hash (w/o Sig)	No	92,507	.02042900	90.5%	22.08
Data Hash (w/ Sig)	No	86,087	.02042260	84.3%	23.72

* Data in Bytes **Cost in Bitcoin

*** Efficiency in Satoshi per Byte of arbitrary data stored

As Fig. 4 and Fig. 5 show, `OP_RETURN` is the most efficient choice for storing small amounts of data (up to 80 bytes). For medium amounts of data (between 80 and 800 bytes), P2FMS is the most cost-effective option, and it provides the least data overhead up to ≈ 10 KB. For large amounts of data (beyond 800 bytes), the Data Drop w/o Sig method provides the least expensive option, and it requires the least data overhead beyond 10 KB. The P2SH-based methods that store data in the input script (Data Drop and Data Hash) have a higher fixed overhead (due to needing an initial transaction to set up the UTXOs that the second transaction redeems), but offer competitive levels of data overhead compared to P2FK and P2FMS for larger amounts of data at much lower costs (since they avoid the burn costs for each UTXO).
Example: for a 50 KB file, the most cost-effective *secure* method (Data Hash w/ Sig) costs approximately 0.012 Ƀ , which is a 61% savings compared to P2FKH (≈ 0.03 Ƀ). At current exchange rates (1 BTC ≈ 2500 USD), this would cost about $30 to publish in the Blockchain.

8. Data Reconstruction

8.1. Methods Involving Burns—All methods relying on fake keys and/or hashes are cumbersome to reconstruct. For P2FKH, each output contains 20 bytes of data to be retrieved, and many ordered outputs can be used to store a contiguous data set. To reconstruct the data, extract the data from the key or hash in each output script.[66] One must be careful to avoid any P2PKH outputs in the transaction that represent "change" addresses; the data outputs are typically marked by their min non-dust values. There does not seem to be a defined limit on the number of outputs a transaction can have.[67] Under the 100 KB size limit, P2FKH has a maximum storage size of 58,680 bytes with a total transaction size of 99,983 bytes. Files larger than this will have to be split among different transactions, and subsequently linked together (either within the Blockchain itself or by external information).[68] This makes fully automatic reconstruction of datasets stored in the Blockchain more difficult. For P2FMS, reconstruction also means avoiding the pushdata OP codes between the fake keys.
8.2. Methods Not Involving Burns—For both Data Drop and Data Hash methods, the data is stored in the input script in the same way. To reconstruct the data, ignore any signature data if present, the pushdata OP codes between the data elements, and the Redeem Script itself. Assuming no malleability concerns, the data will be stored in the same order in which it was broadcast, within a single transaction (up to 100 KB with overhead), achieving a maximum file size of 96,060 bytes.[69] An OP_RETURN output can be used for metadata, such as the name of the file, or the TX ID of the next chunk of data for files larger than 100 KB.[70] To ease retrievability, one may include a single P2FKH output that pays to the hash of the data file being stored, similar to the approach taken by Cryptograffiti.[12] This method allows anyone with this hash to use common blockchain exploration tools to find the transaction where the data was stored. A figure showing the anatomy of an input script is provided, see Fig. 6.

As a point of reference for reconstruction, consider the following transaction, which contains a JPEG image stored using the Data Hash w/o Sigs method:

TX ID: 033d185d1a04c4bd6de9bb23985f8c15aa46234206ad29101c31f4b33f1a0e49
Block: 474586

The Redeem Script data is easily identified as the last data element of each input. The JPEG data precedes the Redeem Scripts, three data elements at a time. The second-to-last input contains only two data elements preceding the Redeem Script. The final input does not contain image data; it is used to pay fees.

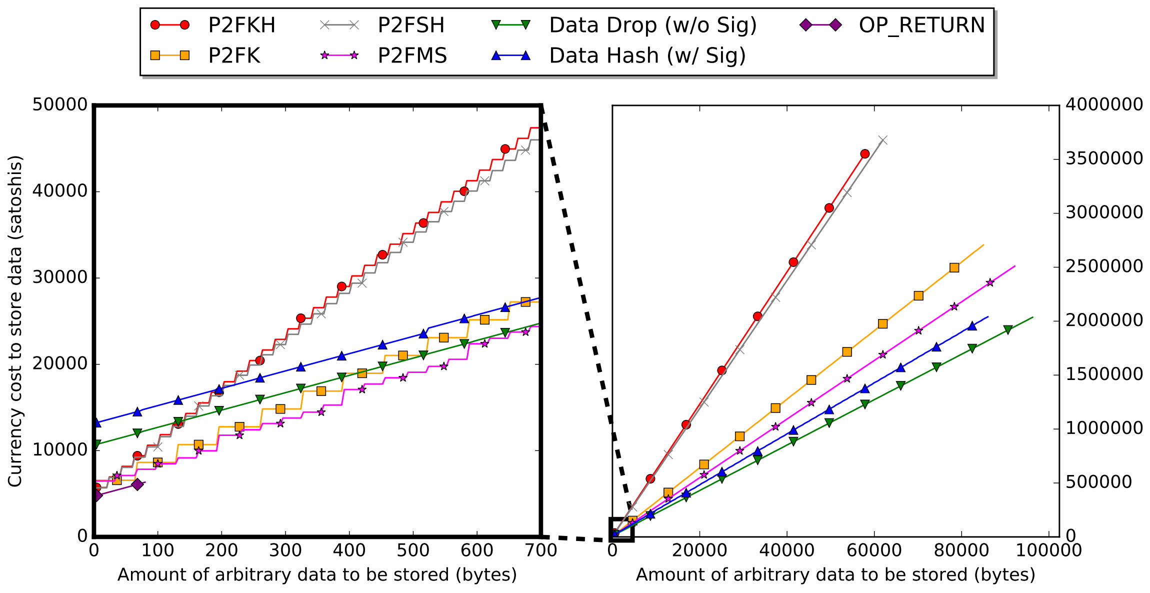

Fig. 4. Total data required vs. stored data size, for small (LEFT) and large (RIGHT) data sizes, up to the maximum size possible within a single transaction.

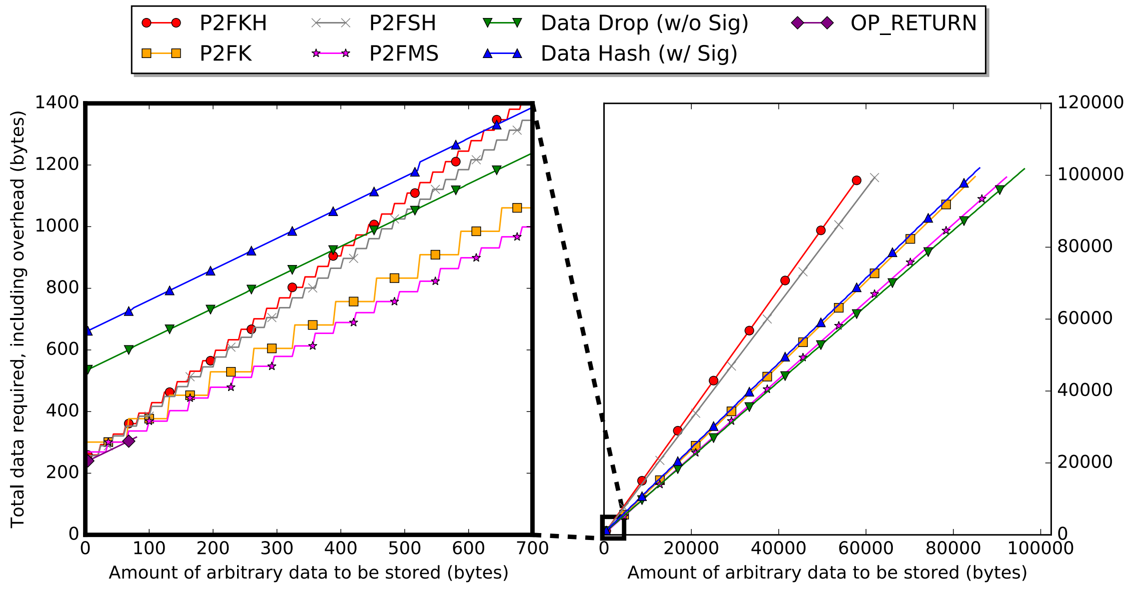

Fig. 5. Currency cost vs. stored data size for small (LEFT) and large (RIGHT) data sizes, up to the maximum size possible within a single transaction. This graph assumes a transaction fee of 20 satoshis/byte and burn values of 1100 satoshis for P2FMS and 546 for other methods requiring burns.

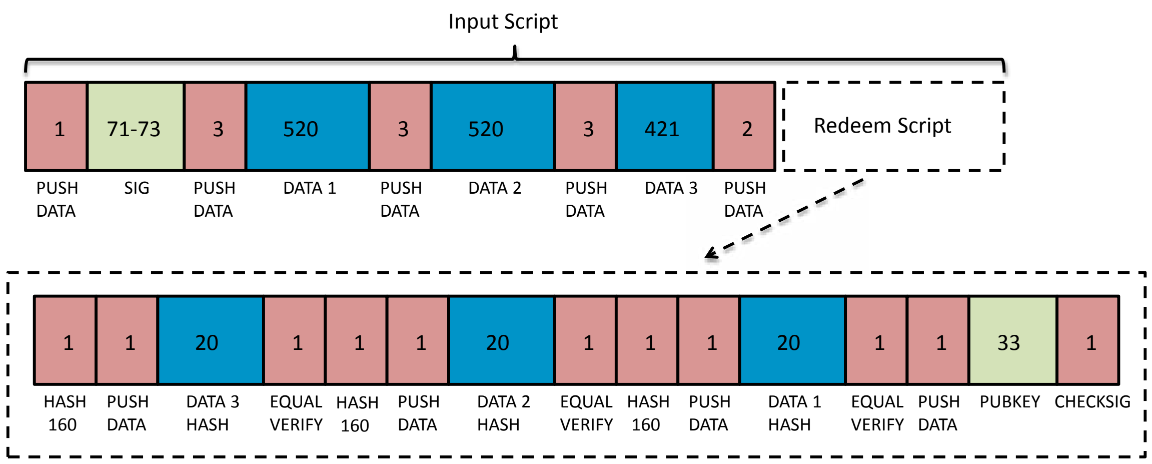

Fig. 6. The anatomy of a maximally-sized input (holding 1461 bytes of arbitrary data) for a Data Hash w/ Sig transaction. It consists of the input script (above) and the Redeem Script (below). The numbers inside the cells label the number of bytes for each field. Red fields denote OP codes, dark blue fields denote the arbitrary data chunks being inserted (and the hashes of that data), and the light green fields denote the signature-related data (omitted for the w/o Sig variant).

9. Conclusion

A comprehensive survey of the benefits and drawbacks of extant methods revealed that there is no optimal data insertion method that dominates all of the others. Instead, different methods will be optimal depending on one's priorities, and the amount of data to store. For small quantities of data, using `OP_RETURN` is a solid choice, and is probably also the closest to an "approved" standard for data publication. For larger amounts of data, if quantity at low cost is paramount and security is unimportant, the Data Drop w/o Sig method may be the best choice. Alternatively, the Data Hash w/ Sig method provides a nice balance of data integrity with an efficient cost function for large data. However, many in the community believe that storing large quantities of data is not an appropriate use of the Blockchain, and that it should be used for storing short hashes of documents (*i.e.* as time-stamped existence proofs) rather than the full documents themselves. Others in the community take a strong free market stance, and hold that if users are willing to bear the cost of data insertion, they should be able to use the technology as they see fit. The purpose of this paper is not to cast value judgments about these perspectives, but rather to encourage *informed* discussion about the technical and economic issues at stake. On a pragmatic level, given Bitcoin exchange rates in recent times, even the most efficient methods may be prohibitively expensive to publish large files, unless the insertion of that data has significant/lasting value to the publisher.

It is striking that P2FKH (which appears to be a dominant approach used by several data publication tools) fares poorly in almost all regards: it creates the most unspendable UTXO bloat, it requires the largest overhead, and it costs the most.[11,12] We have several hypotheses that may explain its (possibly unwarranted) popularity:

(1) It is one of the simplest to implement.[71]
(2) Most people are unaware that more sophisticated approaches (like using input-scripts to store data) exist.[72]
(3) Tool-makers are concerned that more complex methods may be banned in future versions of Bitcoin, which would break compatibility.
(4) Users are concerned that any data that does not create unspendable UTXOs will not be sufficiently permanent, as it may end up being pruned in the future.

This last hypothesis is the most interesting one. On the one hand, as long as Bitcoin survives, surely some nodes will always keep the full and complete ledger (including input scripts), in order to have a complete archive of past transactions, and to be able to verify the hashes of all blocks from the beginning. On the other hand, UTXOs themselves may not be immune to pruning, as the future might bring the possibility of using cryptographic data structures with commitments to store the status of UTXOs without storing the UTXO data directly.[35] However, this would likely serve as a caching optimization for miners/nodes, and the full record including very old UTXOs would still be archived on disk.

As a final caveat, we have attempted to provide a comprehensive review of the major current and past data insertion techniques, but the knowledge and methods contained in this article are based on a scripting protocol that is subject to continual change, and thus some of the methods discussed may become unavailable in the future. For instance, the impact of the Segregated Witness (segwit) BIP on the feasibility of long-term data storage using input scripts is an important question for future testing and research.[34] However, even if future changes to the Bitcoin Core disable or enable new features relating to data storage, there is important academic value in documenting the methods that have been used to date. Knowledge of these methods will be useful for historical research, and may form the building blocks of future methods of data publication for Bitcoin, as well as other cryptocurrencies.

Acknowledgement

We would like to thank members of the Bitcoin Core development team for responding promptly to our questions. We also thank Daniel Zwiener for some interesting conversations. This research was funded by the Augustana New Faculty Research grant program.

Author Contributions

APS directed the project, wrote the bulk of the initial manuscript, and created tables (37.5%), IV wrote code to test insertion methods and search the Blockchain for previously stored data and edited the manuscript (37.5%). FS consulted on technical matters, created graphs, and refined the manuscript (25%).

Notes and References

[1] “The Times 03/Jan/2009 Chancellor on brink of second bailout for banks.” TX ID: 4a5e1e4baab89f3a32518a8831bc87f618f76673e2cc77ab2127b7afdeda33b Block: 0.

[2] Nakamoto, S. “Bitcoin: A peer-to-peer electronic cash system.” (2008) *Bitcoin.org* (accessed July 2017) `https://bitcoin.org/bitcoin.pdf`.

[3] Be aware that anything published in the Blockchain is publicly available and can be accessed by anyone, and it cannot be retroactively altered or removed. For this reason, some use cases may want to store encrypted data or just a hash, rather than plaintext data.

[4] Antonopoulos, A. M. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. Sebastopol, CA: O’Reilly Media 133 (2014).

[5] Textual data is fairly easy to recover, regardless of the insertion method, by searching for ASCII-printable strings, *e.g*. using the Unix *strings* command on the .BLK files. However, binary data files (images, sounds, compressed files, etc.) are difficult to locate without understanding the structure of the transactions and scripts that were used to store the data.

[6] Majakivi, A. (a.k.a. Anduck). *Bitcoinstrings.com* (accessed July 2017) `https://bitcoinstrings.com/`.

[7] Coin Sciences Ltd. Coin secrets (beta) (accessed July 2017) `http://coinsecrets.org/`.

[8] HugPuddle Team. *Bitfossil* (accessed July 2017) `http://bitfossil.com/`.

[9] Shirriff, K. “Hidden surprises in the Bitcoin blockchain and how they are stored: Nelson Mandela, Wikileaks, photos, and Python software.” *Ken Shirriff’s blog* (accessed July 2017) `http://www.righto.com/2014/02/ascii-bernanke-wikileaks-photographs.html`.

[10] Narayanan, A., Bonneau, J., Felten, E., Miller, A., Goldfeder, S. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton: Princeton University Press 217-218 (2016).

[11] HugPuddle Team, embii, The AtomSea. *Apertus* (accessed July 2017) `http://apertus.io`.

[12] Erstu, E. (a.k.a. 1Hyena). *Cryptograffiti.info v0.90* (accessed July 2017) `http://www.cryptograffiti.info/`.

[13] Ron, D., Shamir, A. “Quantitative Analysis of the Full Bitcoin Transaction Graph.” In A. Sadeghi (Ed.), *Financial Cryptography and Data Security, 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, New York: Springer 6-24 (2013) `https://www.springer.com/us/book/9783642398834`.

[14] Reid, F., Harrigan, M. “An analysis of anonymity in the bitcoin system.” In Y. Altschuler *et al*. (Eds.) *Security and Privacy in Social Networks*. New York: Springer 197–223 (2013) `http://www.item.ntnu.no/_media/studies/courses/ttm4546/bitcoin_article.pdf`.

[15] Sleiman, M. D., Lauf, A. P., Yampolskiy, R. “Bitcoin message: Data insertion on a proof-of-work cryptocurrency system.” In *2015 International Conference on Cyberworlds*, IEEE 332–336 (2015) `https://doi.org/10.1109/CW.2015.56`.

[16] Bartoletti, M., Pompianu, L. "An Analysis of Bitcoin OP_RETURN Metadata." arXiv preprint (2017). `https://arxiv.org/abs/1702.01024`.

[17] Miller, A., Juels, A., Shi, E., Parno, B., Katz, J. "Permacoin: Repurposing bitcoin work for data preservation." In *Security and Privacy (SP), 2014 IEEE Symposium on*, 475–490 (IEEE, 2014).

[18] Technically, transactions may also reference other transactions that have not yet been confirmed in a block but are residing in the mempool.

[19] Antonopoulos, A. M. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. Sebastopol, CA: O'Reilly Media 128-9 (2014).

[20] This has changed over time. Check the current Bitcoin Core client to see what is currently allowed as a valid transaction script.

[21] bitcoinj Java Library (accessed July 2017) `https://bitcoinj.github.io/`.

[22] *Blockchain.info* (accessed July 2017) `https://blockchain.info/`.

[23] Chosen because it can handle 15 compressed keys for a multisig transaction as a P2SH, as of Bitcoin Core 0.9.3.

[24] Bitcoin Core Development Team. "Make Transactions with Extra Data in their ScriptSig's Non-Standard." commit message (2012) `https://github.com/bitcoin/bitcoin/commit/39f0d9686095 bce469dbfa52333331a5d15c6545`.

[25] This list of rules is not comprehensive and is subject to change over time. We list only the rules that directly affect data storage methods discussed in this manuscript.

[26] Ironically, due to Bitcoin's decentralized approach, the isStandard method is anything but standard, as miners and nodes on the Bitcoin network can adhere to all, some, or none of these "standard" script restrictions, or even create their own standards to enforce. Moreover, recently the isStandard method has been explicitly moved to a configuration file to be more easily modifiable by users. This means that innocuous-looking scripts that could result in valid execution may never be propagated by a large portion of the network while other strange-looking scripts might be propagated without issue.

[27] BIP 34 adds height as the first item in the coinbase: `https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki`.

[28] Antonopoulos, A. M. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. Sebastopol, CA: O'Reilly Media 187-8 (2014).

[29] TX ID: 930a2114cdaa86e1fac46d15c74e81c09eee1d4150ff9d48e76cb0697d8e1d72 Block: 138725.

[30] An MP3 of Spock saying: "Live long and prosper," spread across multiple transactions inside of block number 345858.

[31] There is some extra data in the first and last transaction that is not essential to reconstruct the image.

[32] The OP code for pushing 65 bytes is 41 which corresponds to 'A' in ASCII, and thus if this method is used for text publication, the first byte should probably be reserved for a newline (or other non-printable ASCII character) so that the popular text extraction scripts do not extract the extraneous 'A' as part of your text.

[33] Nodes could check the x and y coordinates of the public key to ensure they are valid points on the elliptic curve.

[34] Segregated Witness (segwit) Bitcoin Improvement Proposal. `https://github.com/bitcoin/bips/ blob/master/bip-0141.mediawiki`.

[35] Todd, P. "Making UTXO set growth irrelevant with low-latency delayed txo commitments" `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-May/012715.html`.

[36] This table overstates the value of BTC burned to store textual data, since a significant portion (≈ 58 BTC) was paid to the known burn address of all 0s, which likely relates to some proof-of-burn mechanics rather than data storage. However, the table also underestimates both the value burned and the number of transactions that store all data, since binary data (JPEG images, MP3 files, etc.), which tends to be much larger than text, is not accounted for. As of July 2017, there were approximately 52 million total UTXOs (see Lopp, J. "Unspent transaction output set graph," *Statoshi dashboard* (Accessed July 2017) `http://statoshi.info/dashboard/db/unspent-transaction-output-set`), meaning that unspendable ASCII/whitespace/null P2FKH UTXOs comprise about 0.25%. Obtaining an exact accounting of unspendable UTXOs (that store data) is impossible, since arbitrary binary data (especially compressed/encrypted data) can be indistinguishable from legitimate hashes.

[37] Antonopoulos, A. M. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. Sebastopol, CA: O'Reilly Media 133 (2014).

[38] `https://github.com/bitcoin/bitcoin/blob/1ad3d4e1261f4a444d982 a1470c257c78233bda3/src/policy/policy.cpp#L152`.

[39] Using more transactions necessitates more inputs and possibly outputs, increasing fees.

[40] See comments in: `https://github.com/bitcoin/bitcoin/issues/8079`.

[41] More than 3 public keys in a "bare" multisig output script are marked non-standard; higher (*e.g*. 1-of-12) multisigs are generally done using P2SH.

[42] TX ID: a1e537ac06869cf63845ee1fc1a267c5b3bd1db3ac36e6a21fa4ffe20a941b2a Block: 351746.

[43] `https://github.com/bitcoin/bitcoin/issues/8079`.

[44] `https://gist.github.com/gavinandresen/4135d03a56e0ecd146c7`.

[45] Even assuming a very low 5 satoshi/byte fee and TX size of 400, a 2000 satoshi fee is nearly quadruple the size of the current min non-dust value.

[46] HASH160 is a shorthand for RIPEMD(SHA256(Redeem Script)).

[47] BIP 62 requires that the stack contain precisely one non-false value after script execution.

[48] TX ID: afe9034d3afb9d7d8db064b7944d42b30d650d333819cdbe0132ed71febb9725 Block: 475205.

[49] TX ID: d771f31ad04904564da77c1106cde85d06cd641cf2977ffa36f0dd03e89eef4f Block: 307594.

[50] For utmost efficiency, OP_2DROP should be used to drop two data elements from the stack at a time.

[51] TX ID: 94e319d09fc236fb9d7a24e60af8f47ed41ca3cc01e9950c925d806153ed8aa3 Block: 460435.

[52] Sometimes colloquially referred to as the "Western Union" or "WU" method. See Fig. 3.

[53] TX ID: 200f3f6f8a91ae438d1924e5cedca98cea7f0197b9eba11343948b5621ca19ed Block Number: 331804.

[54] This script format resembles the approach used to store textual data in Peter Todd's publish-text.py (see above, n.58), but Todd denied authorship of this transaction, which predates the publication of his Python script by about 5 months.

[55] Likely to remain so, given the current limit on the size of inputs is now only 1650 bytes.

[56] This behavior has been observed in the wild.

[57] The signature is based on a modified copy of the transaction with empty input scripts. See `https://en.bitcoin.it/wiki/OP_CHECKSIG` for details.

[58] Todd, P. publish-text.py (accessed July 2017) `https://github.com/petertodd/python-bitcoinlib/blob/master/examples/publish-text.py`.

[59] A user's transaction may be modified in a minor way that does not affect the transaction's functionality but does alter the TX hash, which can make it more difficult for wallet software to track its status.

[60] Shirriff, K. "Bitcoin transaction malleability: looking at the bytes." *Ken Shirriff's blog* (accessed July 2017) `http://www.righto.com/2014/02/bitcoin-transaction-malleability.html`.

[61] However, this could change if adoption becomes widespread and/or if transaction malleability continues to be a problem in the future.

[62] Note: an adversary may have much stronger motives for mangling the data publication, depending on the data in question. Also, automated sniping scripts may be programmed to act in an economically rational way, but it seems risky to count on this.

[63] Technically, the standard malleability concerns apply regarding altering TX IDs via nonfunctional changes, but the signature protects the integrity of data fields within output scripts.

[64] Although note that if future versions of Bitcoin choose to crack down on data storage using this mechanism, it would be necessary to format your data as a "valid" compressed public key (33 bytes long, starting with 02 or 03).

[65] Fees vary as a result of competition between transactions within the mempool. A transaction fee of 20 satoshis/byte currently appears sufficient to ensure the transaction is included in a block eventually, although in times of heavy traffic it could take a long time (we have witnessed wait times of up to one week with this fee rate). Higher fees increase your transaction's priority, which could be necessary for publication of time-sensitive data.

[66] Do not confuse the PubKeyHash (as bytes of data) with the Base58Check encoded Bitcoin Address.

[67] The largest number of outputs within a single transaction in the Blockchain we found is 13,107. TX ID: dd9f6bbf80ab36b722ca95d93268667a3ea6938288e0d4cf0e7d2e28a7a91ab3 Block: 391204.
This transaction exceeds the 100 KB transaction size limit currently imposed by isStandard.

[68] Multiple transactions all broadcast simultaneously could end up inside a block in any order, or be separated by blocks.

[69] 63.7% more data than P2FKH.

[70] Beware of malleability issues, even a change of a single pushdata OP code will change the TX ID.

[71] In fact, it can be used to store small amounts of text using any standard wallet software, with a relatively simple conversion to convert ASCII data into a Bitcoin address.

[72] We hope this paper will help to remedy this.

Appendix A: List of common terms

We list some common definitions and abbreviations used in the paper.

- **Dust:** If the fees to spend a transaction output (determined from the size of the output and the input required to spend it) would cost more than one third the value of that output, the output value is considered dust. Transactions with dust output values are considered non-standard.
- **Min Non-Dust:** The minimum non-dust value is the least value one can send without the output being flagged as dust. The minimum output value for a P2PKH is currently 546 satoshis. This minimum threshold value changes depending on the script being used.
- **Provably Unspendable:** An `OP_RETURN` UTXO is provably unspendable, meaning that the Bitcoin protocol has marked it as impossible to spend. Thus, it does not need to be included in the set of UTXOs that may be spent in the future. In contrast, some transaction outputs are *effectively* unspendable because their scripts have no known solution. Spending a P2FKH output would require generating a private key that corresponded to that public key, which is astronomically improbable, but does not render the UTXO *provably* unspendable.
- **Redeem Script:** A script that is hashed, and this hash is used as the output of a Pay-to-Script-Hash transaction. The Redeem Script and any inputs it takes are supplied when a user wishes to spend the output that was created. These inputs and the Redeem Script itself are executed and must return true in order for the transaction to be valid.
- **Snipeable:** In this paper, *sniping* refers to the process of re-appropriating the unsigned inputs of an unconfirmed transaction by creating a new transaction with different outputs to hijack the funds those inputs represent. If a transaction has unsecured inputs that can be re-appropriated, it is snipeable.
- **Transaction Malleability:** Transaction malleability refers to the ability to change any part of a transaction without invalidating that transaction. Transaction malleability is often no more than a minor inconvenience, but some of the data storage methods described are subject to a malicious actor potentially changing some or all of the data to be stored.
- **UTXO:** Unspent Transaction Output. Each (non-coinbase) input references a previous UTXO to spend the coins associated with that UTXO. One can think of the set of UTXOs as places where Bitcoin is stored and can potentially be used as sources for future transactions.

Appendix B: Standard Transaction Scripts

(1) Pay-to-Public-Key (P2PK):
Output (locking) Script:`<PubKey> OP_CHECKSIG`
Input (unlocking) Script: `<Sig>`
(2) Pay-to-Public-Key-Hash (P2PKH):
Output Script: `OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG`
Input Script: `<Sig><PubKey>`
(3) Multisig (a.k.a. "bare multisig", or "multisig output"):
Output Script: `M <PubKey 1> ... <PubKey N> N OP_CHECKMULTISIG`
Input Script: `OP_0 <Sig 1> ... <Sig M>`
(4) Pay-to-Script-Hash (P2SH):
Output Script: `OP_HASH160 <RedeemScriptHash> OP_EQUAL`
Input Script: `<Data><RedeemScript>`
(5) OP RETURN:
Output Script: `OP_RETURN <Data>` (up to 80 bytes)
(This output can never be spent, so it has no corresponding input/unlocking script.)

Appendix C: P2SH Storage - Full Scripts

(1) Data Drop w/o Sig:
Input Script:`<Data (520 bytes)><Data (520 bytes)>`
`<Data (520 bytes)><Data (70 bytes)><RedeemScript>`
Redeem Script: `OP_2DROP OP_2DROP <RandomNumber (6 bytes)>`
Note: A random number was included to help prevent the Redeem Script's hash from becoming known. (See section 6.1)
(2) Data Drop w/ Sig:
Input Script: `<Sig><Data (520 bytes)><Data (520 bytes)>`
`<Data (489 bytes)><RedeemScript>`
Redeem Script: `OP_DROP OP_2DROP <PubKey> OP_CHECKSIG`
(3) Data Hash w/o Sig:
Input Script:`<Data 1 (520 bytes)><Data 2 (520 bytes)>`
`<Data 3 (520 bytes)><RedeemScript>`
Redeem Script: `OP_HASH160 <Data3Hash> OP_EQUALVERIFY OP_HASH160`
`<Data2Hash> OP_EQUALVERIFY OP_HASH160 <Data1Hash> OP_EQUAL`
(4) Data Hash w/ Sig:
Input Script: `<Sig><Data 1 (520 bytes)><Data 2 (520 bytes)>`
`<Data 3 (421 bytes)><RedeemScript>`
Redeem Script: `OP_HASH160 <Data3Hash> OP_EQUALVERIFY OP_HASH160`
`<Data2Hash> OP_EQUALVERIFY OP_HASH160 <Data1Hash>`
`OP_EQUALVERIFY <PubKey> OP_CHECKSIG`

Appendix D: BitcoinJ Example Code

Iterating Through the Blockchain

The following code will iterate through the Blockchain up to blk file `END_BLOCK_NUM` (assuming these blk files are stored in ../../Blockchain) and create an output file containing all the output scripts that are not P2PK or P2PKH. This example will process the first 51 blk files (numbered 0-50).

```
// This source code adapted from:
// http://vlkan.com/blog/post/2014/06/27/parse-bitcoin-
blockchain/BriefLogFormatter.init();

final int END_BLOCK_NUM = 50;

// output file to print non-P2PK, non-P2PKH scripts
PrintStream out = new PrintStream(new File("../../output/output0-"
      +END_BLOCK_NUM+".csv"));

// Arm the blockchain file loader.
NetworkParameters np = new MainNetParams();

Context context = Context.getOrCreate(np);
List<File>blockChainFiles = new ArrayList<>();

for(inti=0; i<=END_BLOCK_NUM; i++) {
      String fName = String.format("../../Blockchain/blk%05d.dat", i);
      blockChainFiles.add(new File(fName));
}

// allows us to iterate through blocks
BlockFileLoaderbfl = new BlockFileLoader(np, blockChainFiles);

intblockNum = 0;
// iterate through all blocks

for (Block block : bfl) {

      List<Transaction>txs = block.getTransactions();
      inttxNum = 0;

      // iterate through all transactions
      for (Transaction tx : txs) {
            intoutputNum = 0;
            for (TransactionOutput output : tx.getOutputs()) {
            try { // some transaction outputs cause issues
                  // for example, output script OP_1

            Script script = output.getScriptPubKey();
            // segment script into OP codes
            List<ScriptChunk> chunks = script.getChunks();

// ignore P2PK
if (chunks.size() == 2
      &&chunks.get(0).isPushData()
      && (chunks.get(0).data.length == 65
      || chunks.get(0).data.length == 33)
      &&chunks.get(1).opcode == ScriptOpCodes.OP_CHECKSIG) {
// ignore

// ignore P2PKH
} else if (chunks.size() == 5
      &&chunks.get(0).opcode == ScriptOpCodes.OP_DUP
      &&chunks.get(1).opcode == ScriptOpCodes.OP_HASH160
      &&chunks.get(2).isPushData()
      &&chunks.get(2).data.length == 20
      &&chunks.get(3).opcode == ScriptOpCodes.OP_EQUALVERIFY
      &&chunks.get(4).opcode == ScriptOpCodes.OP_CHECKSIG) {
// ignore

// print other scripts to file
} else {
      out.println(blockNum + "," + txNum + "," + outputNum + ","
      + output.getValue() + "," + script);
}
outputNum++;
} catch(Exception ex){
      // print exceptions to console
      System.out.println(blockNum + "," + txNum + "," + outputNum
            + "," + output.getValue() + ",EXCEPTION: " + ex);
            }
      }
txNum++;
}
blockNum++;
}
```

This can be modified to search for other types of scripts, printable ASCII, etc.

Creating Transactions
Code for creating a P2PKH script to pay to the address corresponding to a key:

```
public static Script createP2PKHScript(ECKey key) {
      ScriptBuilder script = new ScriptBuilder();
      script.addChunk(new ScriptChunk(ScriptOpCodes.OP_DUP, null));
      script.addChunk(new ScriptChunk(ScriptOpCodes.OP_HASH160, null));
      script.data(key.getPubKeyHash());
      script.addChunk(new ScriptChunk(ScriptOpCodes.OP_EQUALVERIFY,
null));
      script.addChunk(new ScriptChunk(ScriptOpCodes.OP_CHECKSIG, null));
      return script.build();
}
```

Other scripts can be made easily by replacing the OP codes here with other codes. Elements that should be added as a pushdata element should be done so with the

`ScriptBuilder.data(byte[] data)` method, as with the PubKeyHash above. An output script created this way can then be added to a transaction in BitcoinJ using the `Transaction.addOutput(Coin value, Script script)` instance method.

Adding inputs to a transaction is less trivial because they must (generally) be signed. Given the transaction `hash` and `index` (within the transaction) of the UTXO to spend, as well as the script to be redeemed, the following code will add an input to a transaction that spends that UTXO. Note, signatures must be generated *after* all of the inputs have been added.

```
public static TransactionInputaddInputToTransaction(Transaction tx,
      long index, String hash, Script scriptPubKey) {

return tx.addInput(Sha256Hash.wrap(hash), index, scriptPubKey);
}
```

Signing an input goes as follows, where `tx` is the transaction, `input` is the input to be signed, `key` is the key used to sign the input, `script` is the output script to be spent, and `index` is the index of the input in the transaction:

```
public static void signInput(Transaction tx, TransactionInput input,
      ECKey key, Script script, long index) {

ScriptBuildersigScript = new ScriptBuilder();
Sha256Hash hash = tx.hashForSignature(index, script,
      Transaction.SigHash.ALL, false);

ECKey.ECDSASignatureecSig = key.sign(hash);
TransactionSignaturetxSig = new TransactionSignature(ecSig,
      Transaction.SigHash.ALL, false);
sigScript.data(0, txSig.encodeToBitcoin());
// Add any additional data/pubkeys/etc. to the SigScript here.

Script scriptWithSig = sigScript.build();
input.setScriptSig(scriptWithSig);
}
```

Other data necessary (such as a public key if redeeming a P2PKH script or arbitrary data to store in the Blockchain) should be added as pushdata elements to sigScript as with the `txSig`.