Contractvm: decentralized applications on Bitcoin

Massimo Bartoletti[1], Davide Gessa[1,2], and Alessandro Sebastian Podda[1]

[1] Università degli Studi di Cagliari, Italy
[2] Helperbit.com

Abstract. We introduce CONTRACTVM, a framework for developing decentralized general-purpose applications on top of the Bitcoin blockchain. Our framework addresses several issues of Ethereum: for instance, it does not need a proprietary blockchain, and it is not affected by attacks which exploit the *verifier's dilemma*. We evaluate the security of CONTRACTVM under different attacker models: overall, we conclude that applications running over our framework are reliable whenever the honest nodes hold the majority of the total hashing power of the Bitcoin network.

1 Introduction

Recently, cryptocurrencies like Bitcoin [24] have pushed forward the concept of decentralization, by ensuring reliable interactions among mutually distrusting nodes in the presence of a large number of colluding adversaries. According to the folklore, Bitcoin would resist to attacks unless the adversaries control the majority of total computing power of the Bitcoin network. Even though the literature reports some vulnerabilities which seem to undermine this belief (see Section 4), in practice Bitcoin has worked surprisingly well so far: indeed, the known successful attacks to Bitcoin are standard hacks or frauds [19], unrelated to the Bitcoin protocol.

Cryptocurrencies leverage on a public data structure, called *blockchain*, where they permanently store and timestamp all the messages exchanged by nodes. Adding new blocks to the blockchain (called *mining*) requires to solve a moderately difficult cryptographic puzzle. The first miner who solves the puzzle earns some virtual currency (some fresh coins for the mined block, and a small fee for each transaction included therein). In Bitcoin, miners must invert a hash function whose complexity is adjusted dynamically in order to make the average time to solve the puzzle ∼10 minutes. Instead, removing or modifying existing blocks is computationally unfeasible: roughly, this would require an adversary with more *hashing power* than the rest of all the other nodes. If modifying or removing blocks were computationally easy, an attacker could perform a *double-spending* attack where he pays some amount of coins to a merchant (by publishing a suitable transaction in the blockchain) and then, after he has received the item he has paid for, removes the block containing the transaction.

The idea of using Bitcoin and its blockchain as the basis for decentralized applications beyond digital currency has been explored by several recent works (see

Section VIII in [9] for a brief survey). For instance, [1,5] design protocols for secure multiparty computations and fair lotteries, and [15] proposes a protocol for Byzantine agreement which is secure when the hashing power of the adversary is strictly less than that of the honest participants. On a more practical side, *Blockstore* [8] is a key-value database with `get`/`set` operations; *Namecoin* [25] is a censorship-resistant domain registration mechanism; *CounterParty* [11] extends Bitcoin with advanced financial operations (like e.g., creation of virtual assets, payment of dividends, *etc.*), by embedding its own messages in Bitcoin transactions. *Ethereum* [10] allows for developing *general-purpose* dapps, interpreted by a decentralized virtual machine which runs over Ethereum nodes.

Before presenting our proposal to decentralize applications through cryptocurrencies, we discuss the feasibility of implementing such an application as an Ethereum dapp [10].

Decentralized applications in Ethereum

Ethereum is a general-purpose cryptocurrency with its own blockchain, which allows clients to outsource computations by embedding them in special blocks called *smart contracts*. These contracts are scripts, written in a special Turing-complete language, which are run by Ethereum nodes upon payment of a reward from the client. When a node completes the execution of a contract, it can claim the reward by broadcasting a transaction with the computed result. Before adding the claimant transaction to the blockchain, and consequently assigning him the reward, the other miners execute a special part of the contract script to verify the correctness of the provided result. Similarly to Bitcoin, invalid transactions (i.e., where the result of the computation does not pass the verification script) can be ignored. In this way, miners are incentivized to verify transactions because, in case one of their transactions in a mined block is invalidated, they would lose the associated fee. Therefore, the correct result of a computation is the one agreed upon by the majority of miners.

Implementing decentralized applications over Ethereum has several issues. First, programmers must write the whole dapp in one of the Ethereum languages, without exploiting existing legacy software or external non-Ethereum services (unless using a trusted oracle). This constraint is required because, while computations carried on by a proper Ethereum virtual machine guarantee to produce correct results, this cannot be ensured for other kinds of computation. Another issue is posed by compromised Ethereum nodes, run by attackers who try to subvert legitimate computations. If a client is not sure to be connected with an honest Ethereum node, he can only protect himself by locally running his own node. This is quite impractical, especially when the computing device has limited resources (e.g., power, bandwidth, disk space): indeed, one has to download the whole Ethereum blockchain, which today requires about 9GB of disk space, and takes about 6 hours to synchronize (the Ethereum blockchain is growing at a rate of 90MB per day). A further practical limitation is that, after a dapp is deployed on the Ethereum network, it cannot be modified anymore;

the only way to update it is to broadcast a new dapp with the modified code, but the old version can still be used.

Besides these practical issues, dapps running over Ethereum are subject to the attacks described in [22]. These attacks exploit the fact that Ethereum miners suffer from the so-called *verifier's dilemma*, according to which they cannot rationally decide whether to verify transactions or not. Whatever choice they make, honest miners are vulnerable of an attack. If a miner honestly follows the protocol by validating all transactions, then an adversary can impersonate a claimant and spam nodes with resource-intensive transactions. Since honest nodes will spend a significant amount of time to verify them, the adversary gains an advantage in the race for mining the next block and obtaining the associated fee. Otherwise, if miners choose to disobey the protocol and skip verification of resource-intensive transactions, then an adversary can claim the reward of a contract by broadcasting a transaction with a meaningless result. Since this transaction will not be verified, the adversary obtains the reward, and in conclusion the client has wasted his money for an incorrect answer.

Contractvm: lightweight decentralization on Bitcoin

We propose and experiment a new, lightweight, technique to decentralize applications on top of cryptocurrencies, which overcomes the drawbacks of Ethereum reported before. We implement this technique into an open-source framework (named Contractvm [17]), which allows developers to write lightweight dapps. Our decentralized applications run on a set of mutually distrusting nodes. The incentive of a node to correctly run a dapp is the same used by other cryptocurrencies: virtual money. Unlike in Ethereum, we resort to the blockchain only to store the stream of update messages sent by clients to nodes. More precisely, we store in the blockchain only *hashes* of these messages, while we use a Distributed Hash Table (DHT) to store the whole messages. Since the number of bits of a hash is limited, we do not need to store them in a proprietary blockchain (unlike Ethereum); rather, we can piggyback these bits directly on the Bitcoin blockchain, by using, e.g., the `OP_RETURN` opcode of unspendable transactions. In this way we achieve two objectives. First, we have a persistent and tamper-proof historical record of all the update messages sent by clients, through which nodes can coherently reconstruct the current state of the dapp. Second, since we only use `OP_RETURN` transactions, whose verification is trivial, miners are not subject to *verifier's dilemma* attacks discussed before.

Correctness of computations is established by a consensus mechanism: responses to a query are sent to a set of nodes which verify the result; a response is considered valid if agreed by the majority of these nodes. Similarly to Bitcoin and Ethereum, this mechanism is secure when the majority of nodes is honest.

Unlike Ethereum, our technique does not impose any constraint on the programming language used to write dapps: developers can invoke legacy applications or external services whenever they need to. Using a consensus mechanism to establish correctness, rather than a decentralized virtual machine as in Ethereum, has an additional benefit: we only need to save in the blockchain the

messages exchanged by clients. In this way we avoid the computational overhead of Ethereum, which must store in the blockchain the whole state of computations. This has also another advantage: each node in our framework can choose which dapp to execute, and so dapps can be updated or removed when needed. To validate new blocks which appear in the blockchain, nodes no longer need to verify the state of *all* dapps.

We establish the security properties of our decentralization technique under various attacker models (Section 4), by considering recent studies on the security of Bitcoin [1,2,3,5,13,15,21,27]. Overall, the results of our analysis establish that the security of lightweight dapps is strictly tight to the security of Bitcoin: vulnerabilities in the Bitcoin protocol induce vulnerabilities in lightweight dapps running upon it. Our analysis also shows that some direct attacks to lightweight dapps are ruled out by design. It may be obvious to some readers that the security level obtained by our decentralization framework comes at a cost in terms of efficiency: to guarantee the immutability of a transaction which updates the dapp state, a client has to wait long enough (in the order of tens of minutes). This is because the probability that an adversary can dismantle such transaction by creating a longer branch in the blockchain drops exponentially in the number of blocks which lie on top of the transaction, and in Bitcoin the average mining rate (i.e., the inverse of the average time needed the solve a cryptographic puzzle) is one every 10 minutes. While this rate may seem unnecessarily low, a recent result by Garay, Kiayias and Leonardos [15] establishes formally (in a core Bitcoin protocol) that this is a necessary precondition in order for an honest hashing-power majority to maintain consistency of the blockchain. A possible way to reduce the latency of transaction confirmation would be to use a cryptocurrency with greater mining rate and fewer active miners (e.g., in Litecoin the average time to mine a block is 2.5 minutes). However, this could be detrimental for the security of dapps, because, compared to Bitcoin, it would be easier for an adversary to control the majority of hashing power. Although implementing dapps on top of cryptocurrencies is not suitable for real-time contexts, this is necessary when the security properties they must enjoy are very stringent.

2 Bitcoin and the blockchain

Bitcoin is a cryptocurrency and a digital open-source payment infrastructure that has recently reached a market capitalization of almost $6 billions [23]. The Bitcoin network is peer-to-peer, and not controlled by any central authority [24]. Each Bitcoin user owns a personal wallet, which consists of a pair of asymmetric cryptographic keys: the public key uniquely identifies the user *address*, while the private key is used to authorize payments. When a user wants to perform a payment, he creates a transaction containing the sender and the recipient's addresses, and a *script* which specifies a validity condition for the transaction (more on this below); then, he broadcasts this "unconfirmed" transaction to the Bitcoin nodes, called *miners*. Miners verify and publish these transactions in the *blockchain*, which essentially implements a *proof-of-work* system [12]. Each miner

maintains a local copy of the blockchain, and a set of unconfirmed transactions received by clients. The goal of a miner is to group transactions into *blocks*, and add these blocks to the blockchain in order to get a revenue.

The mining process. In order to append a new block B_i to the blockchain, miners must solve a cryptographic puzzle which involves the hash $h(B_{i-1})$ of block B_{i-1}, a sequence of unconfirmed transactions $\langle T_i \rangle_i$, and some salt R. More precisely, miners have to find a value of R such $h(B_i \,\|\, \langle T_i \rangle_i \,\|\, R) < \mu$, where the value μ is adjusted dynamically, depending on the current hashing power of the network, to ensure that the average mining rate is of 1 block every 10 minutes. The goal of miners is to win the "lottery" for publishing the next block, i.e. to solve the cryptopuzzle before the others; when this happens, the miner receives a reward in newly generated bitcoins, and a small fee for each transaction included in the mined block (simple transactions which draw coins from one address are usually free of charge). If a miner claims the solution of the current cryptopuzzle, the others discard their attempts, update their local copies of the blockchain with the new block B_i, and start mining a new block on top of B_i. In addition, miners are asked to verify the validity of the transactions in B_i by executing the associated scripts. Although verifying transactions is not mandatory, miners are incentivized to do that, because if in any moment a transaction is found invalid, they lose the fee earned when the transaction was published in the blockchain.

Forks and branches. If two or more miners solve a cryptopuzzle simultaneously, they create a *fork* in the blockchain (i.e., two or more parallel valid branches). In the presence of a fork, miners must choose a branch wherein carrying out the mining process; roughly, this divergence is resolved once one of the branches becomes longer than the others. When this happens, the other branches are discarded, and all the orphan transactions contained therein are nullified.

Scripts. Bitcoin transactions may contain *scripts*, executed by miners while verifying blocks. While most scripts simply verify the transaction signature in order to prevent unauthorized payments, the scripting language features a large set of arithmetic and cryptographic operators. Unlike Ethereum, the Bitcoin scripting language is purposefully not Turing-complete (e.g., it does not allow loops). Since the time for verifying all the transactions in a block (i.e., to execute the associated scripts) is negligibile compared to that required for mining a block, Bitcoin miners are not subject to attacks which exploit the *verifier's dilemma*.

3 Lightweight decentralization on Bitcoin

We now present our lightweight decentralization technique. Our goal is to implement decentralized applications running over a network of nodes which process clients requests. We classify these requests in two groups: *queries* and *updates*. Queries have no side effects on the internal state of a dapp; in contrast, updates may change the state of a dapp.

Note that both queries and updates may be affected by node failures, network partitions, and malicious nodes: e.g., a malicious node may try to cheat by returning a wrong response to a query, or by trying to update the state in an inconsistent way. To cope with these issues, we set up a public ledger through which nodes can reliably reconstruct the dapp state, and we exploit a consensus mechanism to ensure consistency of updates and of query responses. Further, we propose a protocol to prevent the tampering of messages exchanged between nodes and clients.

We use Blockstore [8] as a running example throughout the paper. Blockstore is a key-value database storage with two APIs, `set(key,value)` and `get(key)`. A `set` request saves a new immutable key-value pair in the database, while `get` retrieves a value previously set. Hence, we interpret `set` as an update message, and `get` as a query message.

3.1 Handling update messages

We build upon the Bitcoin blockchain [24], by using it as an *immutable* log of messages. Although the blockchain is primarily intended to trade digital cash, the Bitcoin protocol allows to include a few extra bytes in special transactions (see Section 5). Hence, we do not store full message data in the blockchain, but only their message digests. We use the unique hash of the Bitcoin transaction containing one of these digests as a key for accessing the full message data in a Distributed Hash Table (DHT, see Figure 1). We impose that DHT entries are read-only: nodes can add new key-value pairs, but cannot edit existing ones. Overall, this combination Bitcoin blockchain / DHT gives our public ledger.

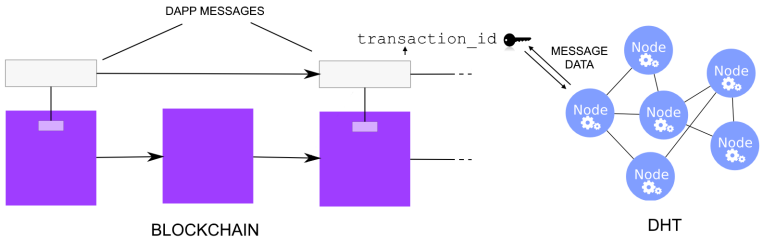

Fig. 1: Schema of the public ledger, with the blockchain and the DHT.

When a node receives an update message it broadcasts it to the other nodes, by publishing the corresponding data in the blockchain/DHT. In this way, we effectively certify and timestamp the updates sent by the clients. However, this is not still enough to prevent a malicious node from publishing an invalid update which compromises the dapp state, and from tampering a client request. We illustrate below a protocol to avoid these malicious behaviours.

1. A sends a raw message m to a dapp node B;
2. B generates an unsigned transaction $t^{uns}_{m'}$ and asks A to sign it;
3. A asks the other nodes in the network to verify if $t^{uns}_{m'}$ is *coherent* with m, and m' is *valid* in the current state;
4. if verification passes, then A signs the transaction with its Bitcoin private key, and sends it to B;
5. B broadcasts the signed transaction $t^{sig}_{m'}$ to the Bitcoin blockchain.

Fig. 2: Message publication protocol between a client A and dapp nodes.

Message validation. Since nodes maintain a local copy of the public ledger, they can always reconstruct from it the current state of dapps (nodes can additionally maintain the state of a dapp in an more efficiency data structure, but this does not affect the security of dapps they run). Similarly to Bitcoin, we do not guarantee that all the messages in the public ledger are valid *ex-ante*; rather, we rule out invalid messages *ex-post*. For instance, in the Blockstore dapp we allow for a public ledger containing `set(k,1)` and `set(k,2)` even though the second message is invalid (because in Blockstore each key can be set only once); the dapp simply ignores the second message. We assume that each dapp has a *validity relation* $q \models m$ which defines when the message m is valid in state q, and a deterministic transition function δ that, given a state q and a message m produces the new state of the dapp. In order to ignore invalid messages, we require that $\delta(q, m) = q$ if $q \not\models m$. Therefore, from a sequence of messages $m_0 \cdots m_n$, a node can reconstruct the dapp state as $\delta(\cdots \delta(\delta(q, m_0), m_1), \cdots)$.

Message publication protocol. We define in Figure 2 the protocol between a client A and dapp nodes, used when A invokes an update request. At flow 2, the node B that handles the client request generates a transaction and asks the claimer to sign it. This transaction need not contain exactly the client message m, but it can contain any message m' *coherent* with m. In this context, coherence means that m' To protect clients from these attacks, at flow 3 we exploit a *consensus mechanism*, through which the client ask the other nodes of the network to validate the transaction and check that is it coherent with the client request. Nodes independently examine $t^{uns}_{m'}$, m and the local state, and reply with a vote. The client collects all votes and then decides whether to sign the transaction or not. We discuss the security of this mechanism in Section 4.2.

Note that the publication protocol prevents malicious nodes from performing tampering and replay attacks. Tampering attacks are unfeasible because modifying client messages would invalidate the signature and the hash; replay attack are ruled out by the Bitcoin protocol, which does not allow to publish two transactions with the same hash.

We remark that an update transaction is not finalized until it is included in a *confirmed* block (see Section 4). Since confirmation involves a delay, it may happen that a transaction appended to the blockchain is later on declared invalid.

3.2 Handling query messages

Queries does not change a dapp state, so their execution does not extend the public ledger. When a client A sends a query to a node B, the node executes the query in its local state, and sends the response to A. To deal with failures and malicious nodes, the client then invokes a consensus call where he asks the other nodes to verify the response of B (which is typically more efficient than computing it). For instance, assume that the current database in Blockstore contains only a pair `{<0,abc>}` when A performs `get(1)`. To do that, A sends the corresponding query message to some dapp node B. The node computes the query response internally, and replies to A with the value (say) `<efg>`. Then, A broadcasts a consensus call to the other nodes, whose majority verifies that the pair `<1,efg>` is not contained in the current state. Hence, A decides to reject the response.

4 Security analysis

In this section we analyse the security of our decentralization technique. First we investigate how known security issues of Bitcoin may affect decentralized coordination models implemented on top of it. Then, we study direct attacks to the nodes of our infrastructure.

4.1 Attacks to the Bitcoin blockchain

Our technique exploits the Bitcoin blockchain to build a trustworthy public ledger to log messages sent by clients to nodes. Although so far Bitcoin has only been affected by standard hacks and frauds, some recent works have spotted some potential vulnerabilities of the Bitcoin protocol, which could be exploited to execute *Sybil attacks* [3] and *selfish-mining attacks* [13].

The Sybil attack is carried out by an adversary that creates multiple fake identities (e.g., different IPs) to simulate the ownership of a large number of nodes in a distributed network. These attacks are usually exploited to quickly propagate malicious information on the network, and to disguise honest participants in a consensus/reputation protocol, e.g. by overwhelming the network with votes of the adversary. The Bitcoin protocol makes Sybil attacks difficult, since the *proof-of-work* system allows to mine blocks/confirm transactions only for *non zero-power* nodes. This implies that an adversary cannot monopolize the currency unless he owns the majority (or near to) of the network hashing power, independently from the number of nodes it controls.

However, if a malicious node uses multiple fake identities, a client may connect, with high probability, to different addresses of the same adversary (and therefore being isolated from the honest network). The attacker itself can propagate information in the network faster than other peers (this can lead to the so-called *double-spending attack* [20,21]). Recent versions of the Bitcoin protocol try to prevent this attack by limiting client outgoing connections to one IP per /16; thus, fake nodes cannot reside in the same network of class C.

In the selfish-mining attack [13], small groups of colluding miners manage to obtain a revenue larger than their fair share in Bitcoin. The attack can be summarized as follows. When a selfish-mining pool finds a new block, it keeps this block hidden to the rest of the network. In this way, selfish miners gain an advantage over honest nodes in mining the next block: this is equivalent to keep a private fork of the blockchain, which is only known to the selfish-mining pool. Note that honest miners still mine on the public branch of the blockchain, and their hash rate is greater than selfish miners' one. Therefore, selfish miners reveal their private fork to the network just before being overcome by the honest miners: recall that the Bitcoin protocol requires to keep mining on the longest chain, in the presence of a fork. Eyal and Sirer in [13] show that, under certain realistic assumptions, this strategy gives better revenues than honest mining: in the worst scenario (for the adversary), the attack succeeds if the selfish-mining pool controls at least 1/3 of the total network hash-rate. Rational miners are thus incentivized to join the selfish mining pool; if the pool manages to control the majority of computational power of the network, the system loses its decentralized nature. Garay, Kiayias and Leonardos in [15] essentially confirm these results: considering a core Bitcoin protocol, they prove that if the hashing power γ of honest miners exceeds the hashing power β of the adversary pool by a factor λ, then the ratio of adversary blocks in the blockchain is bounded by $1/\lambda$ (which is strictly greater than β). Thus, as β (the adversary pool size) approaches 1/2, they control the whole blockchain.

Although these attacks are mainly related to Bitcoin revenues, they can affect the behaviour of any dapp running on top of Bitcoin. In particular, suitably adapted versions of these attacks allow adversaries to trick nodes about the dapp state, forcing them to store locally incorrect values. For instance, the *message revocation attack* exploits the Selfish-Mine strategy and the Sybil attack to convince an honest node to accept an invalid message. We describe an instance of this attack in Blockstore, starting from the empty tuple space:

1. An adversary M, part of a selfish mining pool, sends a message $m =$ `set(0,abc)` to an honest node B;
2. M's pool is mining on a private fork of the blockchain, and the private chain is one block longer than the public one;
3. B broadcasts the transaction t_{B}, containing m, to Bitcoin miners;
4. t_{B} is included in block B_{i+1} of the public chain; B sees the transaction in B_{i+1} and considers it confirmed; so, the local database of B contains only `{(0,abc)}`;
5. M creates a new transaction t_{M}, including the message $m' =$ `set(1,efg)`, and sends it to the selfish mining pool;
6. t_{M} is included in the private block B'_{i+2}, currently hidden to honest miners;
7. selfish miners reveal their private branch; since it is longer than the public branch, block B_{i+1} is discarded, while blocks B'_{i+1} and B'_{i+2} are accepted.

At this point, B believes that the database state is `{(0,abc)}`, while the correct state is `{(1,efg)}` since t_{B} has been discarded. As shown in [13], this attack works also when the selfish-mining pool has no advantage over the honest

miners (i.e., the private chain is not longer than the public one). In fact, they can exploit *Sybil helpers* (fake peers used to quickly propagate information through the network) to convince honest miners to accept B'_{i+1} instead of B_{i+1} (in case of branches with the same length, miners usually mine over the first one they come to know).

Even though such attacks are considered difficult to achieve in practice, nodes can protect from such message revocations by waiting that a transaction is *k-confirmed* before using it to reconstruct the dapp state. Namely, if the last published block is B_n, they consider only transactions which appear in blocks B_i with $i \leq n - k$. This means that an attacker would have to mine at least k blocks to force the revocation of a *k-confirmed* transaction. Rosenfeld [27] shows that, under the realistic assumption that an attacker controls at most the 10% of the network hashing power, fixing $k = 6$ is sufficient for reducing the risk to less than 0.1%. Therefore, we can avoid message revocation attacks by requiring nodes to update their local states only using *6-confirmed* transactions. Note that the value of k can be adjusted to obtain a better responsiveness of the system or, in contrast, a higher security level.

4.2 Attacks to Contractvm nodes and to the DHT

Attacks to the DHT. Recall that, since Bitcoin does not allow to store full message data in transactions, we use blockchain transactions to store message digests, and a read-only DHT to save the full message, whose key is the *transaction id*. Only the node that creates a transaction can write the DHT at that address, and overwriting is not allowed. Under these assumptions, the integrity of the DHT only depends on the security of the hash function used.

Consensus mechanism. Recall that we exploit a consensus mechanism to verify the results of queries, and to check coherence and validity of updates before a client confirms a request. Since this mechanism is majority-based, it is reliable till the presence of attackers in the network is less then 50%, and the distribution of the attackers is uniform (i.e., it is unlikely that a client connects to a set of nodes whose majority is malicious). Further, a *blacklist system* helps clients to reject connections from malicious nodes (e.g., nodes whose responses are repeatedly invalid, or whose consensus votes often disagree with the others). However, an adversary can exploit the Sybil attack shown in previous Section 4.1, to garble the consensus decisions of a victim client: this is done by creating a large number of fake *zero-power* nodes, which do not compute anything but simply return random votes, to force the client to accept an invalid response (or transaction) sent by the adversary. To cope with this attack we adopt the same solution used by Bitcoin, i.e. we limit the client outgoing connections to one IP per /16.

5 Architecture

We now briefly comment the architecture of Contractvm. Its nodes run dapps by processing their requests, (both updates and queries). Nodes keep a local copy

of the state of the dapp, which they update upon reception of relevant messages in the blockchain.

Storage system. The storage system of ContractVM complies to the lightweight decentralized architecture in Section 3. It uses any blockchain that adheres to the bitcoin-core protocol (e.g., Bitcoin, Litecoin, Dogecoin, *etc.*) to provide *proof-of-existence* for the dapp messages. ContractVM nodes exploit the extra bytes available for `OP_RETURN` transactions[3] to store the hash of the message (which points to the whole message in the DHT), and a flag which identifies the dapp delegated to handle the request. Messages are stored in a *Kademlia* DHT (a popular DHT also used by *eMule* and *bittorrent*) run by ContractVM nodes.

Client applications. The client library of ContractVM allows clients to invoke dapp APIs through standard web-based calls (e.g., JSON-RPC). When clients invoke APIs which update the dapp state, they must also publish a suitable transaction in the public ledger, following the protocol described in Section 3.1.

Node daemon. The node daemon follows the schema illustrated in Figure 3. The `Backend` module implements the Bitcoin protocol and it handles new blocks and transactions; when a new block is discovered, the `Chain` module retrieves all the associated transactions, and scans them to extract those messages which contain the framework headers. If the header of message m is found in a transaction with identifier k, the `Chain` module retrieves the content of m from the DHT, using k as key. The content is then checked for validity (hash, signer and size match the data in the DHT), and passed to the `DappManager`, which looks for a suitable dapp to handle the message.

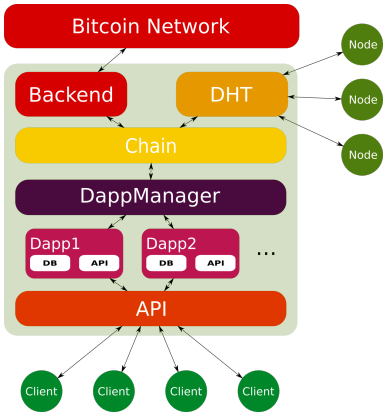

Fig. 3: ContractVM nodes.

6 Validation

To validate the general applicability of our technique, we have applied it to implement some decentralized applications.

[3] In Bitcoin, `OP_RETURN` transactions can be used to store 40 bytes for extra data. In the most recent version of the protocol this limit is 80 bytes; however we prefer to stick to the old limit of 40 bytes for backward compatibility.

Key-value store. We implement a key-value database inspired to Blockstore [8]. The immutability of the blockchain guarantees that a key already set cannot be replaced or deleted by a malicious user. Compared to the Blockstore implementation, which consists of ∼1600 lines of source code, in ours the programming effort is reduced to ∼100 lines. Appendix A shows a detailed walk-through of the code of this dapp.

Linda. We have implemented a vanilla version of Linda [16] featuring `out`, `in` and `rd` operations with data and wildcards. We interpret `out` and `in` as update messages, and `rd` as a query message. Note that the security properties enjoyed by our decentralized implementation are *orthogonal* to those of secure Linda-like languages [6,14,18,28]. These works propose access control mechanisms to prevent untrusted components from interfering with trusted ones in an open network. For instance, these mechanisms protect against denial-of-service attacks where a malicious client repeatedly fires `in(*)` to remove arbitrary tuples from the space, or it overwhelms the tuple space by firing random `out`s. Our decentralized Linda enjoys a different security property, which builds upon the Contractvm consensus mechanism and the immutability of the Bitcoin blockchain: unless a malicious user controls the majority of the hashing power of the Bitcoin network, he cannot tamper the integrity of tuples in the space (which instead is a vulnerability in the above-mentioned proposals). Extending our decentralized Linda with the mechanisms of these secure Linda-like languages just requires to accordingly extend the dapp APIs in Contractvm. Overall, this extension would enjoy the security properties of both realms.

Message queueing. We have implemented a message-oriented middleware (MOM) with the core features of *RabbitMQ* [26], a widespread distributed MOM which allows clients to communicate by sending/receiving messages to/from *FIFO* queues. To deal with node failures, RabbitMQ replicates the queues in various instances, by forming a federation of brokers which act as a single logical broker. Unlike our decentralized implementation, these federated brokers are centralized, because they are still controlled by a single party; were this not the case, a malicious federated broker could act as a Dolev-Yao attacker, by modifying, dropping or rerouting client messages. Note that RabbitMQ cannot give any guarantee about exchanged messages, since their existence and consistency are not (cryptographically) proved. By contrast, our dapp ensures that all messages are correctly dispatched, without assuming any broker to be trusted.

Contract-oriented coordination. Our last case study is a MOM where the interaction among clients is regulated through *contracts*, which formally specify their interaction behaviour [4]. In this setting, clients advertise contracts when they want to establish sessions with other (unknown and untrusted) clients. The coordination layer creates session among users with *compliant* contracts, and it monitors the interaction in each session to detect and sanction contract violations (e.g., when some action is not performed when prescribed by the contract). Therefore, the correctness of such middleware heavily depends on the fact that

the state of sessions is recorded correctly. Centralized implementations, like e.g. the one in [4], have a main drawback: since users must trust a third party to record the state of sessions, an attacker which corrupts the third party may break the correctness of the whole system. Distributed (but not decentralized) implementations have a different drawback: to securely distribute the state of sessions among nodes in the network, suitable cryptographic protocols have to be devised. Our implementation solves both problems: it is decentralized, and the correctness of the session state is guaranteed by the blockchain.

7 Conclusions

We have proposed a general technique which leverages on the Bitcoin blockchain in order to implement decentralized applications on top of Bitcoin, and we have implemented a framework which supports developers in this task without requiring familiarity with the Bitcoin protocol. Decentralization is transparent to developers and clients: they just invoke dapp APIs, without caring about whether the request is served by a peer-to-peer network or by a single node. Applications built upon our framework avoid the classic issues of centralization: e.g., in a centralized architecture clients must rely on a trusted third party, and when this gets damaged or attacked, the system becomes unreliable. Instead, dapps running over Contractvm are reliable whenever the honest nodes hold the majority of the total hashing power of the Bitcoin network.

Our proposal addresses the issues of Ethereum [10] (discussed in Section 1) which, at the time of writing, is the only other tool supporting general-purpose dapps. Unlike Ethereum, in our framework: (i) developers are not bound to a specific programming language; (ii) they can use legacy software or external services; (iii) we do not overload the Bitcoin blockchain, since we only use it to store message digests; (iv) even clients with limited computational resources can use dapps; (v) obsolete/flawed dapps can be updated or even removed (note however that this is considered a *mis*feature in the Ethereum community); (vi) we are not affected by attacks which exploit the *verifier's dilemma* [22]. Overall, we think that these features make Contractvm more suitable than Ethereum for most practical uses. The case studies discussed in Section 6 show that the abstraction layer offered by our framework is usable in practice, and it helps in reducing the programming effort (e.g., in terms of LOC).

Unlike Ethereum, our framework does not provide any incentive to nodes which carry on computations. Although this is not needed for guaranteeing security of dapps, we believe that some form of incentive would make the framework more effective: e.g., we could impose some fee to clients, to be split among nodes which serve their requests; alternatively, we could create a lottery mechanism — similar that of Bitcoin mining — where nodes receive a reward with a certain probability, proportional to the number of dapps they run. The effectiveness of these incentives could be evaluated in a game-theoretic setting, similarly to [7].

References

1. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on Bitcoin. In *IEEE S & P*, pages 443–458, 2014.
2. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. On the malleability of Bitcoin transactions. In *Financial Cryptography and Data Security*, pages 1–18, 2015.
3. M. Babaioff, S. Dobzinski, S. Oren, and A. Zohar. On Bitcoin and red balloons. In *ACM Conference on Electronic Commerce, EC*, pages 56–73, 2012.
4. M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. A contract-oriented middleware. In *Proc. FACS*, 2015.
5. I. Bentov and R. Kumaresan. How to use Bitcoin to design fair protocols. In *CRYPTO*, pages 421–439, 2014.
6. L. Bettini, V. Bono, R. De Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim project: Theory and practice. In *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, pages 88–150, 2003.
7. G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*, pages 142–161, 2015.
8. Blockstore: Key-value store for name registration and data storage on the Bitcoin blockchain. https://github.com/blockstack/blockstore, 2014.
9. J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *IEEE S & P*, pages 104–121, 2015.
10. V. Buterin. Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.
11. R. Dermody, A. Krellenstein, O. Slama, and E. Wagner. CounterParty: Protocol specification. http://counterparty.io/docs/protocol_specification/, 2014.
12. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, pages 139–147. Springer-Verlag, 1993.
13. I. Eyal and E. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454, 2014.
14. R. Focardi, R. Lucchi, and G. Zavattaro. Secure shared data-space coordination languages: A process algebraic survey. *Sci. Comput. Program.*, 63(1):3–15, 2006.
15. J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310, 2015.
16. D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
17. D. Gessa. Contractvm. https://github.com/contractvm.
18. R. Handorean and G.-C. Roman. Secure sharing of tuple spaces in ad hoc settings. *Electronic Notes in Theoretical Computer Science*, 85(3):122–141, 2003.
19. A. Hern. A history of Bitcoin hacks. http://www.theguardian.com/technology/2014/mar/18/history-of-bitcoin-hacks-alternative-currency, march 2014.
20. G. Karame, E. Androulaki, and S. Capkun. Two Bitcoins at the price of one? double-spending attacks on fast payments in Bitcoin. *IACR Cryptology ePrint Archive*, 2012:248, 2012.
21. G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Capkun. Misbehavior in Bitcoin: A study of double-spending and accountability. *ACM Trans. Inf. Syst. Secur.*, 18(1):2, 2015.

22. L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. Demystifying incentives in the consensus computer. In *ACM CCS*, pages 706–719, 2015.
23. CoinMarketCap: Crypto-currency market capitalizations. http://coinmarketcap.com, 2016.
24. S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.
25. Namecoin: a decentralized DNS service. https://wiki.namecoin.org, 2011.
26. RabbitMQ. https://www.rabbitmq.com.
27. M. Rosenfeld. Analysis of hashrate-based double spending. *CoRR*, abs/1402.2009, 2014.
28. J. Vitek, C. Bryce, and M. Oriol. Coordinating processes with secure spaces. *Science of Computer Programming*, 46(1–2):163–193, 2003.

A Dapp source code walk-through

This appendix is a brief tutorial for dapp developers. We illustrate how to write a dapp by commenting the source code of our first case study, i.e. the decentralized key-value storage service. The full source code of this dapp is available at [17].

The key-value storage dapp offers two APIs: `set(key,value)` and `get(key)`. A `set` request saves a new immutable key-value pair, while `get` retrieves a previously set value. Since `set` changes the state of the dapp, this API requires the client to publish a suitable message in the blockchain. Nodes scan the blockchain for new messages, and when they find a `set` request, they save the new key-value pair in their local database. Instead, executing a `get` does not require to publish any message, because nodes can handle this request internally.

The dapp is split in two main parts: the first (Listings 1 to 4) run in nodes, while the second (Listing 5) implements the library that client applications use to interact with the dapp. Note that, while we have chosen Python for developing this dapp, our framework supports arbitrary programming languages. The mechanism used to this purpose is standard: the programmer codes the core features of dapp in her preferred language, and the framework uses them through the foreign function interface.

A.1 Node part

In Listing 1 we define the protocol and the supported messages of the dapp. At lines 1-4 we define a set of constants containing the code for each type of message and the dapp code. Then we extend the `Message` class, by defining a constructor for the `set` message (lines 7-13), and by overriding the function `toJSON()` for the serialization of message data.

The next step is to write the core of our dapp: this is done in Listing 2 by extending the class `dapp.Core`. In this class we define all the methods that interact with the dapp state, including query and pair insertion. We define a function to obtain a value given its key, and another one to set a new key-value pair. We save key-value pairs in the internal database which is automatically created by the framework to store the state of a dapp.

The services offered by the dapp are exposed to client applications as APIs. These APIs are implemented in Listing 3, where we extend the class `dapp.API`, and we create a `dict` object which contains new API calls (lines 5-6). Then, we write our two APIs:

- `set (key,value)`: creates a `set` message with a new key-value pair, and returns message broadcasting information;
- `get (key)`: gets a value for a given key, by invoking the `Core.get` method.

Finally, in Listing 4 we bind all the classes created so far. We use the method `handleMessage` (lines 10-12) to tell the `DappManager` how to handle messages.

```
class BlockStoreProto:
  DAPP_CODE = [ 0x01, 0x02 ]
  METHOD_SET = 0x01
  METHOD_LIST = [METHOD_SET]

class BlockStoreMessage (Message):
  def set (key, value):
    m = BlockStoreMessage ()
    m.Key = key
    m.Value = value
    m.PluginCode = BlockStoreProto.DAPP_CODE
    m.Method = BlockStoreProto.METHOD_SET
    return m

  def toJSON (self):
    data = super (BlockStoreMessage, self).toJSON ()
    if self.Method == BlockStoreProto.METHOD_SET:
      data['key'] = self.Key
      data['value'] = self.Value
    else:
      return None
    return data
```

Listing 1: Blockstore dapp: messages.

A.2 Library

In Listing 5 we define a library module, that binds the API calls described in Listing 3 inside a library, which will be used to write client applications. We do this by extending the `DappManager`. This class includes the services of our dapp, by binding the API calls `bs.get` and `bs.set`. The method `set` only creates and broadcasts a new message containing the given key-value pair; the method `get` performs a consensus query to nodes, and returns the resulting value.

A.3 Example usage

Listing 6 shows a simple "hello world" client of our Blockstore dapp. At lines 4-5 we create a `ConsensusManager`, and we bootstrap it with a seed node. At lines 7 we create a `Wallet` object, by using an external service with private keys saved in the file `app.wallet`. At line 8 we create a `BlockstoreManager`, by using the `ConsensusManager` and `Wallet` objects created before. At lines 10-11 the script asks the user for a key-value pair, and at line 13 it publishes it to the framework. Then, at line 14-15 the script asks the user for a key, and then queries and returns the associated value (if any).

```
class BlockStoreCore (dapp.Core):
  def __init__ (self, chain, database):
    super (BlockStoreCore, self).__init__ (chain, database)

  def set (self, key, value):
    if self.database.exists (key):
      return
    else:
      self.database.set (key, value)

  def get (self, key):
    if not self.database.exists (key):
      return None
    else:
      return self.database.get (key)
```

Listing 2: Blockstore dapp: Core.

```
class BlockStoreAPI (dapp.API):
  def __init__ (self, core, dht, api):
    self.api = api
    rpcmethods = {}
    rpcmethods["get"] = { "call": self.method_get }
    rpcmethods["set"] = { "call": self.method_set }
    errs = { "KEY_ALREADY_SET": {"code": -2, "message": "Already set"},
             "KEY_IS_NOT_SET": {"code": -3, "message": "Is not set"} }
    super (BlockStoreAPI, self).__init__(core, dht, rpcmethods, errs)

  def method_get (self, key):
    v = self.core.get (key)
    if v == None:
      return self.createErrorResponse ("KEY_IS_NOT_SET")
    else:
      return v

  def method_set (self, key, value):
    if self.core.get (key) != None:
      return self.createErrorResponse ("KEY_ALREADY_SET")

    message = BlockStoreMessage.set (key, value)
    return self.createTransactionResponse (message)
```

Listing 3: Blockstore dapp: API.

```
class blockstore (dapp.Dapp):
  def __init__ (self, chain, db, dht, apiMaster):
    self.core = BlockStoreCore (chain, db)
    api = BlockStoreAPI (self.core, dht, apiMaster)
    super(blockstore, self).__init__ (      BlockStoreProto.DAPP_CODE,
                                            BlockStoreProto.METHOD_LIST,
                                            chain, db, dht, api)

  def handleMessage (self, m):
    if m.Method == BlockStoreProto.METHOD_SET:
      self.core.set (m.Data['key'], m.Data['value'])
```

Listing 4: Blockstore dapp.

```
from libcontractvm import Wallet, ConsensusManager, PluginManager

class BlockStoreManager (DappManager.DappManager):
  def __init__ (self, consensusManager, wallet = None):
    super (BlockStoreManager, self).__init__ (consensusManager, wallet)

  def set (self, key, value):
    cid = self.produceTransaction ('bs.set', [key, value])
    return cid

  def get (self, key):
    req = self.consensusManager.jsonConsensusCall ('bs.get', [key])
    return req['result']
```

Listing 5: Blockstore dapp: library.

```
from libcontractvm import *
from blockstore import BlockstoreManager

consMan = ConsensusManager.ConsensusManager ()
consMan.bootstrap ("http://192.168.1.102:9095")

w = WalletExplorer.WalletExplorer (wallet_file="app.wallet")
bs = BlockStoreManager.BlockStoreManager (consMan, wallet=w)

ykey = input ('Insert a key to set: ')
yvalue = input ('Insert a value to set: ')
bs.set (ykey, yvalue)

ykey = input ('Insert a key to get: ')
value = bs.get (ykey)
print (ykey,'=',value)
```

Listing 6: Example usage of the client library for the Blockstore dapp.