Bitcontracts: Supporting Smart Contracts in Legacy Blockchains

Karl Wüst*, Loris Diana*, Kari Kostiainen*, Ghassan Karame†, Sinisa Matetic*, Srdjan Capkun*
*Department of Computer Science, ETH Zurich
†NEC Labs

***Abstract*—In this paper we propose Bitcontracts, a novel solution that enables secure and efficient execution of generic smart contracts on top of unmodified legacy cryptocurrencies like Bitcoin that do not support contracts natively. The starting point of our solution is an off-chain execution model, where the contract's issuers appoints a set of service providers to execute the contract's code. The contract's execution results are accepted if a quorum of service providers reports the same result and clients are free to choose which such contracts they trust and use. The main technical contribution of this paper is how to realize such a trust model securely and efficiently without modifying the underlying blockchain. We also identify a set of generic properties that a blockchain system must support so that expressive smart contracts can be added safely, and analyze popular existing blockchains based on these criteria.**

I. Introduction

Smart contracts, popularized by systems like Ethereum [48], allow nearly arbitrary business logic to be implemented without a trusted third party. Smart contracts are programs whose code and execution results are recorded on the chain. A typical contract enables contract participants to load money to an address or account that is controlled by the contract's code which defines how the loaded money can be later moved out of the contract.

Adding contracts to currencies. While the concept of smart contracts has shown great promise, many currently popular cryptocurrencies, such as Bitcoin [36], Litecoin, Ripple [2] or Stellar [3], do not natively support them. Therefore, it becomes relevant to investigate if contract execution capabilities can be *added* to such blockchains. Since such blockchain platforms have already attracted significant amounts of investment, users and developers, it is often preferable to extend those platforms with contract execution rather than try to migrate the existing users, assets, and investments to other platforms.

Another reason for extending existing blockchains with new contract execution capabilities is the fact that even if some of the existing platforms support contracts, the types of contracts that can be implemented on these systems may be severely limited. For example, Ethereum uses a Turing-complete programming language, but the complexity of computations that can be implemented as contracts are very restricted, due to the built-in block gas limit.

In this paper, our main goal is to design a solution that adds expressive smart contract execution support as a subsystem to existing *legacy* blockchain systems. The primary usage of our solution is to enhance systems like Bitcoin that have no built-in smart contract capabilities. The secondary usage is to extend the contract execution capabilities of platforms like Ethereum that support contracts but have severe limitations on the complexity of allowed computations.

Previous work. Recent research has explored different ways to add contract execution capabilities to blockchains.

For instance, Arbitrum [25] and ACE [50] use *off-chain* execution models, where contract issuers appoint a set of managers who are responsible for executing the contract and communicating the results back to the chain. Hyperledger Fabric [5] uses a similar model in a permissioned setting with an execute-order-validate architecture in which transactions are executed before ordering. The main drawback of such solutions is that they are newly purpose-built systems, and therefore such systems cannot be deployed on legacy systems without modifying the underlying blockchain.

Another proposal, FastKitten [16], relies on *enclaved execution* and *collaterals*, but only supports short-lived contracts that are restricted to known participants. In addition, such a system cannot tolerate enclave compromise. Recently discovered attacks [11], [45], [29], [13], [44] have shown that TEE compromise is a relevant threat. We discuss the limitations of previous solutions in more detail in Section II-B.

Our solution. In this paper, we propose a novel system called Bitcontracts that adds expressive smart contract execution capabilities to legacy cryptocurrencies without requiring protocol changes to the legacy system, and overcomes the main limitations of previous solutions.

The starting point of our solution is an *off-chain* execution model, similar to previous systems like Arbitrum, ACE, or Fabric. In Bitcontracts, the contract issuer appoints a set of service providers that execute the contract's code. The appointed execution set is recorded on the chain together with the contract's code and the contract participants are free to choose if they accept this set. Instead of requiring that all service providers agree on the execution result (as is done in Arbitrum) or trusting the execution environments fully (as is required in FastKitten), we leverage a more flexible *quorum-based trust model* similar to ACE, where execution results are accepted when t out of n service providers report the same result. Such a model can provide both strong security (up to $t-1$ service providers can be compromised) and good availability (up to $n-t$ service providers can be unresponsive).

The main technical challenge that we solve is how to realize such a trust model securely and efficiently without requiring any modifications to the underlying legacy blockchain platform. To achieve this, Bitcontracts leverages the following two ideas. Our first observation is that by storing the state of each contract on the chain, the service providers can remain stateless which reduces protocol complexity and simplifies de-

ployment, since service providers do not need to communicate with each other and do not have to run expensive (in terms of communication) consensus protocols to agree on the current state of the contract which makes our solution efficient. Our second observation is that by binding the validity of each execution result to the latest valid state of the chain, we can enable arbitrary quorum sizes and prevent race conditions where the adversary obtains two acceptable quorums for different execution results affecting the same contract. Due to these observations, Bitcontracts *does not instantiate a new consensus protocol*. Instead, Bitcontracts is purposefully designed such that it guarantees execution integrity and serializability by leveraging the existing consensus protocols of the underlying legacy blockchain. We do this using an *execute-order* model, which ensures (at ordering time) that only serializable transactions can be included in the chain. This is a crucial difference to the *order-execute-validate* architecture of Hyperledger Fabric, which requires an additional validation step that cannot be retrofitted into a legacy blockchain without protocol changes. We discuss this in more detail in Section II-B.

Bitcontracts requires no changes to the underlying legacy blockchain, as long as it supports four generic properties. The first property is *auxiliary storage* which is needed to store contract state on the chain. Auxiliary storage is possible by encoding data to legacy transactions. The second is *collective authorization* which is supported as multi-signature transactions by most blockchains. The third is *state dependency* which ensures serializability in our solution. State dependency is implicitly supported in all UTXO-based systems and can be explicitly enforced in most account-based systems. The fourth property is *transaction atomicity* which enables contracts to perform complex operations safely.

We analyze popular cryptocurrencies, including Bitcoin, Litecoin, Zcash, Ethereum, Ripple, and Stellar and show that these properties are supported in most popular blockchain deployments. In few cases, when one of the properties is missing, we explain how they could be easily added.

Finally, we analyze Bitcontracts in three ways. First, we explain how contract execution can be incentivized in a solution like Bitcontracts through standard means like execution fees and subscription models. Second, we prove that Bitcontracts provides strong safety and liveness guarantees under our chosen, flexible trust model. And third, we implement a prototype of Bitcontracts that runs Python contracts on top of unmodified Bitcoin and other legacy cryptocurrencies and show that the involved transaction fees are small (e.g., few USD cents per contract call). We also evaluate the transaction sizes and costs for Bitcontracts in comparison to Ethereum, by crawling data from 130k Ethereum blocks and leveraging the contract call transactions from the 100 most popular Ethereum contracts. We conclude that running popular smart contracts in Bitcontracts is practical — and often even cheaper than in Ethereum.

Contributions and roadmap. To summarize, in this paper we make the following contributions:

- *New solution:* We propose Bitcontracts that enables secure, efficient and expressive smart contracts on unmodified legacy cryptocurrencies (Sections III & IV).
- *Requirement analysis:* We identify the minimal set of properties that a blockchain needs to provide to allow expressive smart contracts and analyze the existing blockchains based on this criteria (Section V).
- *Bitcontracts analysis:* We explain how incentives can be added to Bitcontracts (Section VI); we prove that Bitcontracts provides safety and liveness (Section VII); we provide a Bitcoin-compatible implementation of Bitcontracts (Section VIII); and we evaluate the execution costs of Bitcontracts on popular blockchain platforms as well as an analysis of the costs of executing popular real-world smart contracts in Bitcontracts (Section IX).

II. Problem Statement

In this section, we motivate our work and explain the limitations of previous solutions.

A. Motivation

Blockchain technology has gathered significant business interest that is largely focused on smart contracts and their applications (see Appendix A for brief background).

Three basic options for deploying smart contracts exist: The first is to use an existing blockchain platform like Ethereum that provides built-in contract support. The second is to create a new blockchain platform. And the third option—which we investigate in this paper—is to *retrofit* contract execution capabilities to an existing and unmodified legacy blockchain.

There are several reasons to enhance existing platforms with new contract execution capabilities. The first reason is that platforms like Bitcoin have already gathered significant investment and user base. For instance, at the time of writing (June 2020), the market cap of Bitcoin is more than half of the entire blockchain market [1]. Migrating all the invested funds and existing users to a new platform is expensive and complicated.

The second reason is that successfully launching a new blockchain platform is hard. A fully-functional blockchain platform requires an entire ecosystem, including developers, tools, miners, investors, users, clients and more. Bootstrapping all of this from scratch is very expensive and likely to fail.

The third reason is that existing blockchain platforms with smart contract support have significant restrictions on the types of computations that can be implemented. For example, the *gas* limits of Ethereum restrict contracts to very simple and short computations. In many business use cases, it would be desirable to run more complex computations than what is allowed by Ethereum currently.

And fourth, the existing smart contract platforms are based on dedicated (often niche) programming languages. For example, Ethereum in practice requires the use of Solidity or Vyper that can be compiled to EVM bytecode[1]. Most developers are more familiar with general-purpose languages

[1]Creating compilers that compile other languages to EVM-bytecode is, of course, possible but would require significant engineering effort for each language.

like Python or Java. Developers would benefit if they could use their favorite programming language for writing smart contracts and if the same contract code could be re-used across different smart contract platforms.

Given these reason, our main goal in this paper is to *add expressive smart contract execution capabilities to existing legacy blockchains*, with a secondary goal of enabling developers to write contracts in their favorite programming language. We focus on enabling *Ethereum-style smart contracts* which is probably the most common definition of the term "smart contract". In Appendix A, we discuss how Ethereum-style smart contracts compare to other types of on-chain computations such as ones that operate on private data.

B. Limitations of Previous Solutions

Side-chain execution. One known approach to extend legacy currencies with additional functionality is to use a *side-chain*. Several proposals for side chain mechanisms exist, targeted at different use-cases. For example, Liquid [19] is targeted at enabling fast asset transfers, but does not provide expressive smart contracts. Rootstock (RSK) [30] enables smart contracts for Bitcoin using a side chain that is based on its own currency (RBTC) that is pegged to the value of a Bitcoin. This is achieved by issuing an amount of RBTC only when the same amount of BTC was previously locked under a multisig condition to a threshold set of trusted parties. Smart contracts can then be run on the RSK side chain and perform payments using RBTC. Side chains – no matter their purpose – generally require trust in a *fixed set* of parties (or even a single trusted party) [41], instead of allowing contract participants to accept trust assumptions on a per-contract basis. This means that only contracts whose users trust the same set of parties can co-exist on the same side chain and interactions between contracts on separate chains is not possible. In addition, in terms of usage, a side chain is equivalent to moving funds to a separate blockchain system except for the (usually) fixed exchange rate between the currencies of the two systems. Users that want to use a contract on the side chain first need to move funds to the side chain and wait for enough confirmations to pass before they can use the contract. After the execution they then need to move the funds back to the main chain if they want to hold their funds on the main chain. This requires a total of five transactions for a single contract execution (two each on the main chain and side chain for moving the funds back and forth, plus one for contract execution).

Off-chain execution. Another approach is to run contract code *off-chain* in few chosen execution nodes. Arbitrum [25], ACE [50], and Yoda [17] follow this approach. In Arbitrum, the contract issuer appoints a set of "managers" who are responsible for executing the contract. Once a contract call is complete, the managers send the execution results to miners who accept them only if all managers report the same execution result (otherwise the system falls back to an expensive dispute resolution protocol). Since contracts are executed decoupled from the consensus process, systems such as ACE, Arbitrum, and Yoda enable execution of complex contracts without slowing down the consensus process. However, the main drawback of such solutions is that they require changes to miners and thus such solutions cannot be deployed to legacy blockchains without modifications to the blockchain protocol.

State channels [33], [21], [22], [32] constitute another approach to move on-chain execution of smart contracts off the chain. However, such constructions require fallback mechanisms and joining procedures that rely on on-chain execution. Thus, state channels are limited to blockchains that already support expressive smart contracts.

Execute-order-validate model. A specific variant of off-chain execution is the *execute-order-validate* model that is used in Hyperledger Fabric [5]. Fabric is a popular permissioned blockchain system, where a contract-specific set of *endorsers* execute transactions independent of the consensus process. Executed transactions are sent to an *ordering service* that establishes a total order on them and assembles them into blocks that contain the readsets and writesets (i.e., state changes) of the contract execution. After that, blocks are broadcasted to *peers* (roughly speaking, peers correspond to system participants like miners). To ensure transaction serializability, Fabric requires a validation step in which, for each transaction, peers sequentially check the values stored in the readset of the transaction and check if they are still the same as the values in the current state of their local ledger. Otherwise, the transaction is invalidated and its state changes are not applied.

Such a validation step is necessary to ensure serializability in Fabric. Since the endorsers execute transactions before they are ordered, they execute them based on the latest committed state. Namely, it is possible that two transactions, Tx_A and Tx_B, are received by the endorsers at roughly the same time and thus executed based on the same state. To consider a simple example, assume that the contract contains a state variable x ($x = 0$ before the execution of both transactions) and both transactions increment this value by one. Both read/writesets now contain a read for $x = 0$ and a write for $x = 1$ in the endorsements. After ordering Tx_A before Tx_B, for example, we have a write $x = 1$ for Tx_B, instead of write $x = 2$, as it should be when executing Tx_A and Tx_B sequentially. The validation step solves this problem by invalidating Tx_B.

The execute-order-validate model is suited for new deployments of a blockchain system where the base protocol can dictate that all peers (i.e., system participants like miners) perform transaction validation. Our goal in this paper is to add smart contract execution capabilities as a subsystem that operates on top of an unmodified legacy cryptocurrency. In this respect, the execute-order-validate model is not suitable for such a subsystem, since if the read-/writesets validation is only performed by the subsystem participants, then money transfers in the subsystem are not consistent with the rules for money transfers in the legacy cryptocurrency.

We illustrate this problem with an example. Consider again two conflicting transactions, Tx_A and Tx_B, in the same block. Assume that the transactions were created by a subsystem that runs on top of unmodified Bitcoin. Now consider that before executing these two transactions the contract has a balance of

1 coin, and Tx_B sends this coin to some other party. Given the execute-order-validate model, Tx_B will be invalidated by subsystem participants, even though it has been included in a block. That is, from the point of view of the subsystem participants, the contract still has a balance of 1 coin that it can use in future transactions. However, all participants of the legacy cryptocurrency will adhere to the Bitcoin protocol and they do not invalidate this transaction. Thus, from their point of view, the account associated with the contract has a balance of 0. Any transaction sent by clients of the subsystem that would cause the balance of the contract to decrease, would thus be rejected by these other parties, including miners, even though they would be valid within the subsystem.

Enclaved execution. The next known approach is to outsource contract execution into *trusted execution environments* (TEEs) like SGX enclaves. Ekiden [14] is an example system that follows this approach. The main problem with such solutions is that if the adversary compromises the enclave where the contract is executed, he can arbitrarily violate its integrity and, e.g., steal all the contract-controlled funds. Recent research on SGX side channels [11], [45], [29] and micro-architectural attacks [13], [44] has shown that TEE compromise is a practical threat that should be considered.

Blockchain multiparty computation. Recent research has also explored how to run secure multiparty computation (MPC) on blockchains. The main goal of such works is to improve *fairness* of existing MPC protocols, rather than adding contract execution to legacy blockchains, but such schemes can also be seen as specific types of smart contracts.

In MPC, a set of parties provide *private* inputs and jointly evaluate a function over them. A common challenge is that malicious parties can stop participating once they learn the function output and prevent other parties from learning the output and thus violate fairness. An impossibility result from Cleve [15] proves that no MPC protocol can be fair without an honest majority. In recent research, it has been shown that this fairness problem can be alleviated, to an extent, using blockchain. Andrychowicz et al. were the first to show how to implement fair 2-party lottery on Bitcoin [6]. This result was extended to n-party lotteries [9], playing poker [28] and other MPC protocols [26]. In such schemes, each party must place a deposit on the blockchain. If a participant stops participating, he loses his deposit (i.e., these systems create monetary *incentives* against fairness violation but cannot completely prevent it).

If such MPC protocols are treated as smart contracts, they have several functional limitations. First, these solutions are customized to very specific computations and extending the same ideas to arbitrary business contracts and applications is hard. Second, all contract participants and the duration of the contract have to be known in advance which is not true for many smart contracts in systems like Ethereum. And third, some of these solutions require modifications to the underlying blockchain, such as adding new instructions to the scripting language [28].

Enclaved multiparty computation. A recent work called FastKitten [16] combines techniques from Ekiden [14] and blockchain-based MPC [6], [9], [28], [26] to enable contract-like computations on top of unmodified Bitcoin. Similar to Ekiden, FastKitten also uses an SGX enclave to execute the smart contract. Similar to MPC schemes, all participants must place a deposit in the contract before its execution. In addition, the operator of the TEE has to post a deposit that equals the sum of all user deposits. If the protocol fails (because one user misbehaves), all parties except the misbehaving get their initial deposit back.

From a functional point of view, FastKitten has the same problems as MPC schemes (contracts must have fixed participants and limited lifespan) and the contracts enabled by FastKitten are therefore much more restricted than in Ethereum which allows an unlimited lifespan and a dynamic participant list. FastKitten also has security problems. One example is an attack where multiple participants collude. For example, if it becomes clear from an execution up to the last round that Bob and Charlie will lose all of their deposit to Alice, the first two can collude such that Bob stops sending messages. While Bob will still lose his deposit, Charlie will receive his full collateral back and Alice is cheated out of her gain. Thus, smart contracts in FastKitten are not completely self-enforcing under malicious behavior. Finally, FastKitten is vulnerable to TEE compromise similar to Ekiden.

III. Bitcontracts Overview

In this section, we provide an overview of our solution Bitcontracts. First, we describe our execution model and discuss the challenges of realizing it. After that, we explain the main ideas of Bitcontracts and define common properties that a blockchain must provide to support it.

A. Execution and Trust Model

The starting point of our work is an *off-chain execution model* in which the execution of contracts is decoupled from the consensus process. An obvious approach is to distribute trust among several service providers, such that one trusts a set of service providers collectively, as is done in system like Arbitrum [25], Fabric [5] or ACE [50]. Service providers in such a model could be reputable companies or non-profits. In Bitcontracts, we follow this approach as well.

However, unlike Arbitrum, where all service providers must unanimously agree on the contract execution results, we adopt a *more flexible trust model* similar to ACE and Fabric, in which the contract creator can choose the requirements for acceptable execution results per contract. Namely, the creator of a contract chooses a set $\mathcal{E}$ that consists of n service providers and a threshold t of required authorizations. A state transition caused by contract call is considered valid if the transaction committing the results is authorized by at least t members of the executing set $\mathcal{E}$. Contract participants are free to take part in contracts only if they agree with the chosen specification, i.e., they agree with the assumption that fewer than t members of $\mathcal{E}$ are malicious. Note that, in order to

ensure safety and liveness at the same time, a majority of the service providers in $\mathcal{E}$ need to be honest, even if the threshold t is lower. Otherwise a malicious majority could sign wrong results (if $t \leq n/2$) or violate liveness (if $t > n/2$).

However, it is still valuable that t can be freely chosen to be able to prioritize either safety or liveness. Such a model allows *flexibility* depending on the requirements of the use case. For example, if strong integrity is required, but high availability is not crucial, one may choose a large $\mathcal{E}$ with t close to $n = |\mathcal{E}|$. If on the other hand, $\mathcal{E}$ is chosen such that all of the members are trusted and high availability is required, one can choose a low threshold such as $t = 1$. Contracts that are expected to be active for a long period could specify conditions for replacing service providers within the contract itself, as discussed in [50].

Our trust model modifies the typical trust assumptions of smart contract systems slightly. In Ethereum, the specification of a smart contract is defined by its code (cf. Appendix A). In our system, the specification also includes a set of service providers and the threshold. Importantly, all users decide if they trust and agree with this specification. They can make this decision by either performing the due diligence themselves or by trusting other parties to perform it for them, similar to checking the trustworthiness of contract code in Ethereum, but they are not required to trust the contract creator. Finally, also similar to Ethereum, they only need to trust the specification of contracts they participate in and are not affected by the execution of other contracts. For example, if one contract's executing set is compromised, other contracts remain secure.

B. Challenges

The primary technical challenge that we solve is how to realize the above outlined execution and trust model securely and efficiently for contract execution on legacy blockchains. Next, we discuss why simple solutions fail to solve the problem.

Where to store state? We start by considering the storage of a contract's state. The first possible option is to store the state of each contract *off-chain* at the service providers. Due to our quorum-based execution authorization, not every service provider needs to be involved in every contract call, and thus some service providers might not have the latest state of the contract. Therefore, in this approach, the service providers would need to run a consensus protocol between them to ensure consistency of the contract's state. This is a costly process, adds unnecessary overhead to the service providers, and incurs restrictions on the size of the quorum as it needs to be more than $\frac{2}{3}n$ given n service providers. While the consensus process could be simplified by storing the hash of the current state on the blockchain (and thus partially leveraging the blockchain consensus), the service providers still need to ensure that all of them are in possession of the latest state. If the system should be able to include multiple transactions involving the same contract in the same block, they still need to ensure consistency and thus coordinate to ensure that they do not end up in diverging states.

The second option is to store the state of each contract *on-chain*, i.e., publish it on the blockchain of the underlying cryptocurrency. This option leverages the consensus mechanism of the underlying cryptocurrency, instead of requiring that the service providers need to run an expensive and complicated[2] consensus protocol separately and ensures that all parties have access to the latest state. This also allows clients to individually verify the correctness of every execution result. Another advantage of storing the full state on-chain is that service providers themselves can remain completely stateless and do not need communication with other service providers or the blockchain, i.e. they only need to communicate with the client and do not need persistent storage for contract state.

How to ensure consistency? It may seem that storing the state on the blockchain is sufficient to ensure consistency between the service providers and thus the integrity of the smart contracts they execute, but this is not the case. We illustrate this with a simple example attack.

Assume an idealized blockchain where transactions cannot be reorganized and every created block is final. Also assume that the contract's issuer sets the authorization threshold to $t = \frac{2}{3}n$ and the adversary controls $\frac{1}{3}n$ of the service providers. The adversary triggers two different contract calls to two distinct sets of honest service providers, sized $\frac{1}{3}n$ each. Both sets authorize the contract call based on the current state of the contract that is stored on the chain. The adversary then authorizes both contract calls with the $\frac{1}{3}n$ service providers that he controls, and thus both contract calls have the required $t = \frac{2}{3}n$ authorizations. Then, the adversary publishes the first execution result that updates the contract's state and, for example, transfer funds out of the contract. After that, the adversary publishes the second execution result that updates the contract's state based on the previous stale state which means that the results of the first contract call are reverted, except for their side effects such as money transfers.

A simple solution to this problem would be to mandate that the threshold t must always be sufficiently large to prevent such attack, i.e., $t > \frac{2}{3}n$. This simple solution has two problems. First, it prevents deployments where low thresholds should be used for best possible availability. And second, it would not prevent the above outlined attacks in blockchains, where temporary forks are possible (e.g., all legacy blockchains based on PoW consensus).

In Fabric and ACE (which have a similar trust model to Bitcontracts and also leverage off-chain execution), this problem is solved in different ways. ACE uses an *order-execute-commit* model and Fabric uses an *execute-order-validate* approach. Both systems therefore require additional steps after ordering the transactions. However, ordering in legacy cryptocurrencies has side-effects, such as money transfers, as discussed in Section II-B, and thus potential inconsistencies need to be resolved before or during the ordering phase. One of our challenges is, therefore, how to ensure transaction serializ-

[2]Deploying consensus protocols requires thousands of lines of non-trivial code that is difficult to check for correctness [23].

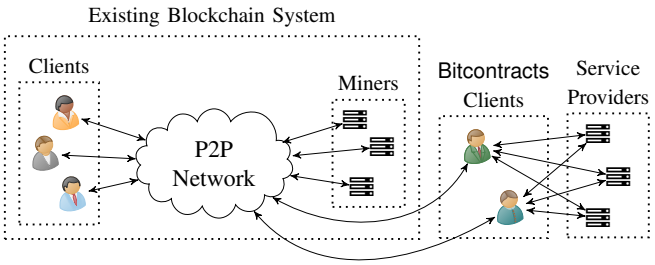

Fig. 1: **Bitcontracts overview.** Bitcontracts extends existing blockchain systems without changing their protocol, i.e. existing nodes such as clients and miners are agnostic to Bitcontracts. Bitcontracts clients interface with the blockchain and Bitcontracts service providers. Service providers are stateless and do not need to interact with the blockchain system.

ability both in the subsystem (that adds contract execution capabilities) and the legacy cryptocurrency without requiring additional steps after ordering. In other words, our solution needs to be compatible with an *execute-order* architecture.

C. Overview of Bitcontracts

Next, we explain the main ideas behind Bitcontracts, and introduce the properties (**1-4**) that are required from a legacy cryptocurrency to enable it. Figure 1 shows an overview.

Bitcontracts combines *off-chain execution* of contracts with *on-chain storage* for contract state. This design decision allows the service providers to be stateless, enables flexible trust models and high availability, provides transparency towards the contract's clients, and most importantly *does not require a new consensus protocol* since it leverages the consensus of the underlying legacy blockchain instead. We acknowledge that storing the contract state on the chain comes with a cost (that we evaluate in Section IX), but argue that these benefits combined outweigh this drawback.

In Bitcontracts, a smart contract account is a normal blockchain account managed jointly by multiple service providers using **(1) multiparty authorization** like multi-signature transactions. The current state of each smart contract is stored on the chain using another common feature of blockchains, **(2) arbitrary data storage**.

Because the contracts' state is recorded on the chain, the contracts' clients can assemble the latest contract state from the chain at any time. For each contract call, the client that initiates the call assembles the contract's state and sends it to the service providers that are registered for this contract together with the contract's code and call input parameters.

The service providers execute the contract call and encode the execution results as a signed state update transaction that they return to the client. The client completes the transaction by combining the received signatures from t service providers that report the same result so that the required multiparty authorization is fulfilled and broadcast the completed transaction to the P2P network. The miners accept the state change transaction if it is signed by at least t service providers who control that contract's account.

We note that it might seem counterintuitive to have the client assemble and broadcast the final transaction, as he can then choose not to broadcast it if the results are unfavorable to him. However, in Bitcontracts this design decision does not provide the client any advantage, since contract execution is deterministic (see Section VIII-A for details) and therefore the client can know the results of the contract execution before initiating the contract call. This means that instead of creating the contract call and then withholding the result, the client could simply calculate the result himself and then choose not to call the contract. The same property holds in other smart contract systems such as Ethereum.

To prevent the attacks described in Section III-B where the adversary obtains two valid quorums for conflicting contract states, in Bitcontracts we require that the contract's state used as input in a contract call is always the latest on-chain state of the called contract. Such enforcement is possible if the validity of a transaction can be conditioned on the current state of the blockchain, a property that we call **(3) state dependent transaction validity**. Such a referencing mechanism is available in many existing cryptocurrencies, for example, in UTXO-based cryptocurrencies, transactions reference UTXOs that must be outputs of previous transactions which have not yet been used as inputs in a transaction. Since transaction validity, and therefore state dependent transaction validity, is checked by miners, this mechanism can be leveraged to prevent attacks or benign race conditions, where a contract call is executed based on an old state, even if the blockchain experiences short-lived forks and at the same time allows usage of arbitrary quorum sizes.

This idea of *binding off-chain execution results to on-chain state validity checks* that are performed by the miners of the underlying legacy blockchain is a key feature of Bitcontracts. It enables Bitcontracts to guarantee execution integrity for contracts that are executed off-chain without it having to implement a separate consensus protocol of its own (see Section VII). The same idea also allows Bitcontracts to be compatible with an *execute-order* architecture, in which, similar to Hyperledger Fabric [5], contract calls are first executed and then ordered. In contrast to Fabric which uses an *execute-order-validate* approach (cf. Section II-B) this notion, however, removes the separate validation step. Recall that in Fabric, this validation step is needed, because the blockchain consensus mechanism only performs ordering of transactions without performing any checks and thus all peers need to check for conflicts in the read-/writesets of the contracts to ensure serializability. In Bitcontracts, such conflicts are prevented during the mining process (ordering step), since miners check for state-dependent transaction validity, which ensures that all values read during the contract execution correspond to the values from the previous state.

Finally, Bitcontracts enables contracts where a single transaction performs multiple separate money transfers. This is possible, when the underlying blockchain supports **(4) atomic multitransactions**, i.e., transactions that atomically execute payments from multiple sources to multiple recipients.

D. Cryptocurrency Properties

Above we introduced informally properties **1-4** that the underlying cryptocurrency must provide to support Bitcontracts.

These properties are necessary to support *our* execution and trust model, securely and efficiently, on unmodified legacy blockchains. We do not claim that these properties are necessary or sufficient for *every* contract execution system. For example, if a different trust model with a single executing node is chosen, fewer properties may be sufficient.

Next, we specify these four properties more precisely in the format of interfaces. This allows us to keep our system design in Section IV agnostic of the underlying blockchain platform. Note that these interfaces are defined on *transactions* and they can be used without direct access to the blockchain itself, if the relevant transactions are supplied to the service providers by the client. In Section V, we analyze how these properties are supported in existing, widely-used cryptocurrencies.

(1) Multiparty authorization. To allow a distributed set of service providers to perform state transitions for a contract, the cryptocurrency must support a form of multiparty authorization, i.e. a mechanism that allows a set of n entities to collectively authorize a transaction with signatures from a threshold number t of them. An example of such authorization is multi-signature outputs in systems like Bitcoin. This ensures that changes to the smart contract state are only committed to the chain, if enough service providers authorized the state transition. The threshold is set per account, i.e. if funds are being transferred from multiple sources, each of them may have their own threshold that needs to be met.

We abstract authorization for a transaction Tx as an interface $\sigma = \texttt{sign}(\mathsf{Tx}, sk)$, where sk is a secret key of the authorizing entity and a transaction Tx is valid if the threshold condition is met for every source of transferred funds. To verify authorization on a transaction for an account, miners and other nodes use a predicate $\texttt{verify}(\mathsf{Tx}, \Sigma, PK, t)$ where $\Sigma = f(\sigma_1, \ldots, \sigma_t)$ is some function[3] on a list of signatures, $PK = (pk_1, \ldots, pk_n)$ is the list of public keys and t is the threshold value associated with the account.

(2) Arbitrary data storage. The cryptocurrency must allow storing auxiliary (non-financial) information in a transaction in order to support stateful contracts with stateless service providers. Storing the contract state on chain ensures that all contract participants receive the latest state and are able to continue interacting with the smart contract. An example for this property is the ability to store data in Bitcoin scripts.

For a transaction Tx we abstract appending some data d to this storage as an interface $\mathsf{Tx}.\texttt{append\_data}(d)$ and reading as $d = \mathsf{Tx}.\texttt{read\_data}(loc, len)$, where loc specifies the location and len specifies the length of the data to read.

(3) State dependent transaction validity. As the service providers should remain stateless, the transaction validity rules of the cryptocurrency must allow the validity of a transaction to be conditioned on a state references in the transaction. That is, the transaction should reference a previous transaction to be valid if and only if that previous transaction has been included in the chain and resulted in the currently valid state. In Bitcoin and similar currencies, this is trivially supported through the UTXO model, since a transaction is only valid if all inputs are outputs of a previous transaction (i.e. included in the chain) and have not been spent (i.e. represent the current state). In Section V we discuss how this property is provided in account-based systems. In addition to enabling stateless service providers, this property prevents time-of-check to time-of-use (TOCTOU) problems, because every new state can directly reference the previous one and base its validity on it (see Section VII for details).

For a transaction Tx, we abstract this condition as an interface $\mathsf{Tx}.\texttt{require\_previous}(id)$ where id is a unique identifier for a state or previous transaction and where Tx will only be accepted as valid if id refers to the most recent associated state or transaction.

(4) Atomic transactions. A smart contract should be able to receive and send funds within a smart contract call. This necessitates that atomic transactions with multiple origins and multiple destinations must be possible, i.e. the smart contract should be able to receive and send funds in a single contract call. In UTXO-based cryptocurrencies this can simply be done by creating a transaction that uses UTXOs from different parties as inputs and creating multiple outputs. In other cryptocurrencies, one atomic transaction may require creating multiple transactions for which atomicity is guaranteed through other mechanisms (see Section V). Note that rolling back a transaction due to permissionless consensus mechanisms such as proof-of-work does not violate transaction atomicity, since the transaction is either rolled-back in full, or not at all. Therefore, all UTXO-based cryptocurrencies, as well as many others, support this property independent of their consensus mechanism.

For a transaction Tx, we abstract this property as an interface $\mathsf{Tx}.\texttt{add\_transfer}(src, dest, val)$ that adds a transfer of funds with value val from src to $dest$ to the transaction. If a transaction contains multiple transfers, this interface is called multiple times. All fund transfers are then executed atomically.

IV. Bitcontracts Specification

In this section, we describe the Bitcontracts system in detail. We start with our system model, and then explain the contract deployment and execution.

A. System Model

There are three types of entities in Bitcontracts, as shown in Figure 1:

Existing Blockchain System. Bitcontracts extends existing blockchain systems with smart contracts. Existing entities such as blockchain clients and miners (or stakers in Proof-of-Stake systems), as well as the P2P infrastructure are agnostic to Bitcontracts and thus do not need to be modified.

Bitcontracts Clients are participants and creators of smart contracts. They connect to the blockchain's P2P network and to service providers for contracts in which they are participating. Bitcontracts clients can create smart contracts by creating

[3]This can for example be the identity function, which would be the case in Bitcoin multisignatures. However, this could also be some form of signature aggregation such as BLS [10] signatures.

a transaction that sets an initial state and initial funds for the contract and specifies the responsible service providers and broadcasting this transaction to the blockchains P2P network.

Service Providers. A set of service providers called *provider set* ($\mathcal{P}$) that can execute smart contracts. Service providers are stateless and do not necessarily need to connect to the blockchain. Service providers get requests from clients to execute a contract based on a given state, execute this contract and send the result back to the client. Each provider creates a keypair for receiving and sending transactions on initialization and publishes the public key. This can be done in several ways; a provider can publish it on the blockchain, he can make it accessible on some publicly available website, or he can send it to clients directly.

B. Contract Deployment

Smart contracts in our system consist of a piece of code written in an arbitrary language, some funds and a contract state stored on the blockchain as a key-value store, which allows for easy retrieval of the state during contract execution. The smart contract account can be viewed as an account managed by a quorum of service providers that can collectively authorize transactions.

In order to deploy a smart contract, the client chooses an executing subset $\mathcal{E} \subseteq \mathcal{P}$ of an arbitrary size n and a t-out-of-n trust model that describes which number t of the providers out of the set $\mathcal{E}$ have to attest to the correctness of smart contract execution. Note that this set is collectively trusted for contract execution results and thus the chosen service providers are typically known (and not anonymous) entities. The set of service providers responsible for the execution of the contract is part of the contract specification and is thus specific to a contract and not to a transaction. The client then creates a transaction Tx whose recipient is a new account that is managed by $\mathcal{E}$ collectively, i.e. a subset of $\mathcal{E}$ of size t can authorize transactions from this account. For example, in UTXO based currencies, this would correspond to a t-out-of-n multisig output. In addition to any initial funds going to the contract account, this transaction contains a hash of the contract code, a hash of its initial state, and the initial state itself in its auxiliary storage. This is added to the transaction by the client before broadcasting using the `append_data` interface. The client then broadcasts the transaction and makes the code available to any other party that should be able to interact with the smart contract. If the contract should be publicly available, he could even publish the contract code in the contract creating transaction as well. Alternatively, he can publish it on some publicly available website.

C. Contract Execution

To execute a smart contract, a client has to contact at least t of the n providers in the contract's executing set $\mathcal{E}$ to execute the smart contract. If one of the contacted providers does not respond, he needs to contact an additional one. A sequence diagram for the contract call and execution is shown in Figure 2.

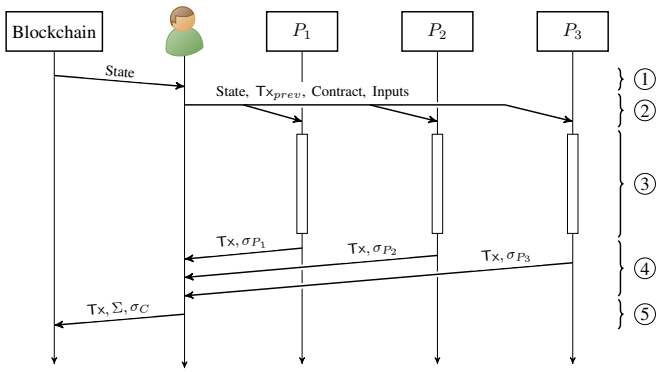

Fig. 2: **Contract call.** To call a smart contract, the client first assembles the state from the blockchain and then sends the state, the previous transaction, and his inputs to the service providers. The service providers then execute the contract call and send the resulting transaction as well as their signatures to the client, who finalizes the transaction and broadcasts it.

① The client first fetches the current state of the contract from the blockchain by going through the contract's past transactions and assembling the state from all state changes stored in them. This does not need to be repeated in full every time; clients can continuously update their local state, or they can even rely on a service that provides them with the most recent state (which they can check using the state hash stored in the most recent transaction), similar to current lightweight blockchain clients.

② To each of the contacted providers, the client then sends the current state, the previous contract transaction Tx_{prev}, the smart contract code, any inputs for the smart contract execution, and any information required to send funds from the client to the smart contract (e.g. UTXOs from the client). Service providers can also store the code, but the design described here allows service providers to be fully stateless.

③ Each provider P_k then proceeds as follows:

(i) The provider computes the hash of the contract code, retrieves the hash of the contract code from Tx_{prev} using the `read_data` interface and compares the two values. If the values match, he continues, otherwise he aborts.

(ii) The provider does the same for the state, i.e. he retrieves the state hash from Tx_{prev}, compares it to the computed hash of the state received from the client, and aborts if the values do not match.

(iii) Given the state, parameters, and additional inputs, the provider executes the smart contract. This contract execution can change the state of the contract and can initiate transfer of funds to other addresses.

(iv) The service provider creates a raw transaction Tx and makes it dependent on the previous transaction using Tx.`require_previous`$(\mathsf{Tx}_{prev}.id)$.

(v) The provider hashes the new state and appends the hash of the contract code as well as the state hash to the transaction using the `append_data` interface.

(vi) The provider computes the list of state changes from the previous state to the new state, serializes this list and appends it to the transaction storage using the `append_data` interface.

(vii) If the smart contract receives funds from the client or the execution causes funds to be transferred to another address, the service provider uses the `add_transfer` interface to add the transfers to the transaction Tx.

(viii) Finally, the service provider P_k creates a signature on the transaction as $\sigma_{P_k} = \texttt{sign}(\mathsf{Tx})$, and sends the transaction and signature back to the client.

④ The client receives the transactions Tx as well as a signature σ_{P_k} from each provider P_k. Since the contract execution is deterministic, each of the providers creates the same transaction and provides a signature over it. The client then assembles the multiparty signature Σ from all received signatures $\sigma_{P_1}, \ldots, \sigma_{P_t}$. If the client is sending funds to the contract (or is providing funds to pay for fees), he also provides his own signature σ_C on Tx.

⑤ Finally, the client broadcasts the signed transaction $(\mathsf{Tx}, \Sigma, \sigma_C)$ and it can be included in the blockchain.

D. Contract Dependencies

For contracts calling other smart contracts, we need to ensure that (i) the whole call (including subcalls) is executed atomically, and (ii) that execution integrity is guaranteed given the chosen trust model of each contract. This requires that state changes for all contracts are committed with a transaction (or sequence of transactions executed atomically) signed by a quorum of the executing set of each involved smart contracts.

In order to execute a contract call with subcalls, the client first runs the contract call locally to determine the set of involved contracts and then sends the state, the latest transaction, and the code of all involved contracts to all service providers, together with the inputs to the contract call. The service providers then perform the same steps as listed above in Section IV-C, checking the code and state hashes for every involved contract and executing the full call chain. Since the resulting transaction can only be included in the chain if it fulfills the multisignature condition of all involved contracts, this ensures that all state changes are only applied if all of the quorums are reached.

E. Use of Oracles

Because of the way, Bitcontracts is built, service providers can natively act as oracles for many use cases since the contract code can directly connect to external websites or data feeds. For example, if Alice and Bob set up a contract to bet on whether it will rain on new year's eve and they both trust the same feed for weather data, they can write a contract in which they both lock some funds such that: the contract directly accesses this feed using https, checks if weather data is available for new year's eve, and then pays out to the winner of the bet. This contract can then be called on new year's day by the winner of the bet. Since the result of accessing this feed and checking whether it rained should be the same independent on which honest service provider this is executed, no external oracle is needed.

TABLE I: **Required Properties** supported by popular cryptocurrencies. (✔ = provided property, ○ = partially provided property, ✖ = not provided property).

Model	System	Storage of Auxiliary data	Multiparty Authorization	State dependent Tx validity	Atomic Transactions
UTXO	Bitcoin	✔	✔	✔	✔
	Litecoin	✔	✔	✔	✔
	Zcash	✔	✔	✔	✔
	Dash	✔	✔	✔	✔
	Cardano	✔	✔	✔	✔
	Monero	✔	○	✔	✔
Account	Ethereum	✔	✔	✔	✔
	Ripple	✔	✔	✔	○
	Stellar	✔	✔	✖	✔
	EOS	✔	✔	✔	✔

V. Property Analysis

In this section we analyze popular blockchain systems and explain how they provide the properties that Bitcontracts needs. Table I summarizes our analysis.

A. Storage of Arbitrary Data

Some account based cryptocurrencies, such as Stellar and Ripple, offer explicit mechanisms to store arbitrary data. Others, such as Ethereum and EOS, support this through their smart contract system, as arbitrary data can simply be sent as a parameter to a contract call. In Ripple, this is supported using a `Memos` field that adds data to a transaction, while Stellar allows writing to a key value store of the account using a `Manage Data` operation.

Most Bitcoin forks support specific outputs that only store data using the `OP_RETURN` instruction. This allows only a small amount of data to be stored per transaction since at most one output using this instruction can be used per transaction. There exist several workarounds for this that allow storing more data per transaction with some overhead [42] for currencies supporting Bitcoin script. We explain one of them in Section VIII-B. In general, arbitrary data can usually be stored in transaction fields reserved for addresses, as addresses generally resemble a random string and have no constraints that can be checked. For example, in Monero, to store more than 32 bytes of data (which can be stored as payment id) in a transaction, one has to create dummy outputs that store the data in the field for the stealth address. This has quite a large overhead, however, since it requires a range proof of 2kB [40] for every 32 bytes of data.

B. Multiparty Authorization

A mechanism for multiparty authorization is often desired, e.g. for wallets from companies, to enhance security and therefore usually supported in cryptocurrencies. Most UTXO-based cryptocurrencies, such as Litecoin, Zcash, and Dash, are forks of the Bitcoin protocol and also support Bitcoin script, which enables multisignatures. Note that "multisignature" in this context refers to transaction authorization that requires individual signatures from multiple parties. It does not refer to multisignature schemes that usually require a protocol between signers to collectively produce one single signature. Even though Cardano is not a fork of Bitcoin, it also supports script and allows for multisignatures. Stellar and Ripple (using the account model) implement multisignatures differently, but

still support them, while Ethereum and EOS already support expressive smart contracts that can and have to be used to implement multiparty authorization.

Monero is a special case since it does not explicitly support multisignature accounts. Instead, multiparty signatures have to be created by splitting keys among multiple parties and running an interactive signing protocol. In addition, they are not well supported in software which makes them cumbersome to create [4], [31]. While this is enough to be compatible with our system, it requires that the service providers interact between each other, instead of just communicating with clients.

C. State Dependent Transaction Validity

In the UTXO-model, state dependent transaction validity is an *implicit* consequence of the model, as inputs to a transaction must be unspent outputs of a previous transaction, which makes validity of a transaction directly dependent on the previous transaction. All UTXO-based cryptocurrencies therefore support this property.

In account based cryptocurrencies, such policies have to be supported *explicitly*. This is the case in Ripple, for example, that provides a special mechanism that allows specifying the hash of the previous transaction from an account in the `AccountTxnID` field of a transaction. The transaction will then only be accepted by the ledger if this value is the hash of the latest transaction of that account. Such a mechanism is missing in Stellar, but could easily be added the same way, to enable support for Bitcontracts.

As another possibility, state dependent transaction validity can be implemented within an existing smart contract system, as is the case with Ethereum and EOS, by creating a smart contract that stores the most recent state hash and only accepts state updates if the previous state referenced in the update corresponds to the stored value.

D. Atomic Transactions

As with state dependent transaction validity, all UTXO-based cryptocurrencies support atomic transactions as an implicit consequence of the model: transactions must support multiple inputs and outputs, since otherwise transactions could not have variable amounts. In currencies supporting smart contracts, such as Ethereum and EOS, this is again supported through the smart contract system.

Stellar allows adding multiple payments to a transaction. If the payment is from a different source than the sending account, the respective account also needs to sign the transaction. Ripple does not natively support atomic transactions with multiple sources and destinations, but native support could easily be added, similar to Stellar. Even without native support, a form of atomic payments can be added on top of Ripple using the PathJoin protocol [35]. However, to use this protocol to ensure transaction atomicity, service providers would need to interact with each other and would need to keep track of the ledger state.

VI. Incentives

The Bitcontracts specification, described in Section IV, implicitly assumes that service providers will execute client transactions. In this section, we explain how transaction execution can be incentivized and how service providers and clients can be protected from each other. We focus on transaction fees that have become a common way of incentivizing work in blockchain systems, but we emphasize that service providers could also be incentivized through other (off-chain) means such as subscription fees similar to cloud computing services.

In addition to explicit incentives added through transaction fees, misbehavior is also disincentivized implicitly through loss of reputation, since clients will generally choose known, reputable entities and not anonymous parties. Bitcontracts provides undeniable evidence of misbehavior, such as signing false state transitions, which would damage the service's reputation. However, explicit negative incentives, such as financial penalties for misbehavior require enforcement through some form of trusted party. In a legacy system without native smart contracts, such a trusted party is not available. Negative incentives could be enforced within a Bitcontracts contract itself, but would not be very useful, since they would then rely on the trust model not being violated (in which case there is nothing to gain from misbehavior).

As shown Figure 1, Bitcontracts supports deployments where the service providers are totally disconnected from the blockchain. In such deployments incentives must be handled off-chain (e.g., subscription models). When on-chain incentive mechanisms like transaction fees are used, service providers need to be able to check the blockchain.

Introducing fees. A straightforward application of fees to Bitcontracts system would be as follows. At the time of calling a Bitcontracts contract (step ② in Section IV-C), the client specifies a fee that he is willing to pay to the service providers. The client includes funds to pay this fee, separately for each service provider, to the contract call request. If the service providers find the included transaction fees acceptable, they execute the contract call, include the fee payment from the clients funds in the same transaction, and return the signed result to the client (steps ③ and ④). By signing the Bitcontracts transaction and publishing it on the legacy blockchain (step ⑤), ownership of the fees is effectively transferred from the client to the service providers.

Such fee mechanism is simple and efficient to deploy, as it adds no additional latency to contract call processing. However, it has two minor drawbacks. The first drawback is that a malicious client could simply not sign and publish the final transaction (i.e., skip step ⑤) and thus cause unpaid contract call execution for the service providers. Clients do not gain any benefit from such misbehavior and thus rational clients do not have an incentive to abuse service providers like this. Similar to other DoS and resource exhaustion attacks, service providers could defend themselves through known mechanisms like asking clients to solve cryptographic puzzles during at times of heavy load [20], [24], [7] or by requiring

slightly higher fees overall to compensate occasional unpaid execution work.

The second drawback is that some service providers could choose to "free-ride" and not execute the call, in the hope that other service providers will complete the work and they get still paid. This drawback could be addressed by contract creators who could select reputable entities as service providers.

On fairness. An ideal transaction fee mechanism would provide fairness between service providers and clients. The service providers would perform the contract call execution work only if they are guaranteed to receive the fee. The clients would only pay if the know that service providers will execute and sign the contract call. This problem is close to the notion of *fair exchange* [38], where a seller releases a product, like an output of some computation, to a buyer only when he is are guaranteed to receive a matching payment.

Unfortunately, the existing fair exchange protocols are not applicable to our setting for two main reasons. First, the existing legacy-chain compatible fair exchange protocols like Zero-Knowledge Contingent Payments (ZKCP) [8], [47], [12] protect a digital "product" like a computation result, but they do not protect the task of *computation itself*. Thus, if we would adopt one of the existing ZKCP protocols, a malicious client could still impose unpaid work on service providers. Second, the existing fair exchange protocols usually consider a 1-to-1 setting with a single buyer, while our execution model has n sellers (i.e., n service providers).

Because the classical notion of fair exchange is not applicable to our setting, we target a slightly different goal. Our goal is to provide execution incentives for service providers and at the same time protect them from malicious clients (as well as possible). More precisely, service providers should be incentivized to execute transaction quickly, free-riding should be discouraged, and clients should not be able to impose execution work on service providers without a fee payment.

Example incentive mechanism. Next, we describe an example incentive mechanism that achieves these goals. The client pays the transaction fees to a t-out-of-n multisig account that is controlled by the service providers collectively. This payment can be done on-chain (or through a payment channel in the case where the client interacts often with the smart contract). The service providers then check before executing the contract if they received the payment. If this is the case, they proceed as usual.

After some fixed time interval (e.g., a month), the service providers *collectively* create a transaction from their multisig account that pays out a share to each service provider in proportion to the number of contract transactions (that appear on the chain) with a signature from the respective service provider. Since the service provider check that they received the payment before they execute the contract call, they are guaranteed to be paid collectively, and since fewer than t of the service providers can be malicious, the fees are guaranteed to be distributed based on the fraction of contract transactions to which the service providers contributed. This also incentivizes fast responses from service providers, since they are more likely to be included on chain and clearly disincentivizes free-riding.

Such an example incentive mechanism has two minor drawbacks. First, it increases contract call processing latency slightly, since service providers need to wait until they have received the payment before they execute the contract. Second, a malicious client can still favor some service providers over others by selectively choosing which signatures are included to the final transaction. We consider the development of incentive mechanisms that provide perfect fairness for one buyer of computing work and a quorum of sellers an interesting open problem that is beyond the scope of this paper (and potentially of independent interest).

VII. Security Analysis

In this section, we analyze Bitcontracts in terms of its safety and liveness guarantees.

Safety. The main safety or security condition that a contract execution system like Bitcontracts needs to provide is that every contract is executed correctly. We say that a system provides *execution correctness* for a particular contract if all calls to that contract that appear in the chain are serializable and each call provides control-flow integrity based on the resulting state of the call immediately preceding in the serialization of all calls. Based on this definition, we make the following claim:

Claim 1. *Given the specification of contract A, which defines an executing set $\mathcal{E}_A$ that consists of n_A service providers and quorum size t_A, the following holds: If fewer than t_A service providers from $\mathcal{E}$ are compromised, the contract provides execution correctness, i.e. serializability and control-flow integrity.*

Proof. We consider the following cases:

1. Correct client inputs. Assuming that the contract code and state provided by the client are correct, all honest service providers will only sign a transaction if the contained state transitions are correct, i.e. the new state is the correct result of the smart contract execution, given the state they received as input. It follows that if fewer than t_A service providers are compromised, a transaction containing false state transitions cannot gain a quorum for contract A and thus cannot be committed to the blockchain, i.e. we have execution integrity based on the state provided by the client.

2. False previous state or contract provided by the client. For this case, we assume that the client provides the correct previous transaction Tx_{prev}. Even though we know from above that the state transitions themselves must be correct, they are based on a state and contract code provided by the client. The client could therefore send a forged state as input state. However, the previous transaction Tx_{prev} that led to this state contains the hashes of the state and the contract. The service providers check that the provided state and contract correspond to these hashes and abort if this is not the case, i.e. a state transition based on a mismatched state or contract code cannot reach a quorum for A.

3. False or outdated previous transaction provided by the client. The above does not take into account that the client could also provide a forged or outdated previous transaction Tx_{prev}. A transaction can be outdated even in the absence of an attack, simply because two clients call the contract at approximately the same time. However, our system needs to ensure no state transitions based on such an outdated state are committed to the chain to prevent race conditions and specifically TOCTOU vulnerabilities. Before signing the resulting transaction Tx and sending it back to the client, the service providers condition the validity of the new transaction Tx on Tx_{prev}, i.e. Tx will only be accepted if Tx_{prev} was committed to the blockchain, and it was the most recent transaction of the contract account. This ensures that even though a transaction based on an outdated (or false) previous transaction may reach a quorum, it cannot be committed to the blockchain since the validity criteria, which are checked by miners, are not fulfilled.

It follows that a transaction with a quorum of signatures must provide control-flow integrity, directly references a single valid previous state, and if it is accepted into the chain, is the only such transaction referencing this state, which ensures a unique serialization. Since a contract call that involves multiple contracts requires a quorum for each involved contract and the validity of the final transaction is based on the previous states of all involved contracts, the above applies to all contract calls independent of whether they involve other contracts or not and independent of client behavior, i.e. even if a client misbehaves or colludes with malicious service providers. □

Finally, we note that our system does not have any security implications on parties that are not participating in a contract, even if said contract has a malicious quorum. This follows directly from the fact that Bitcontracts does not change the protocol of the underlying cryptocurrency.

Liveness. The main liveness condition that a system like Bitcontracts should ensure is that every transaction from an honest client that does not conflict with another transaction (i.e. one including the same contract and based on the same state) is executed and can be committed to the blockchain. Based on this definition, we make the following claim:

CLAIM 2. *Given a contract call T from an honest user involving k contracts C_i ($1 \leq i \leq k$) that define executing sets $\mathcal{E}_{C_i}$ with quorum t_i, the following holds: If the contract call does not conflict with other contract call and an honest quorum of size t_i is reachable in $\mathcal{E}_{C_i}$ (for all $1 \leq i \leq k$), liveness is guaranteed for transaction T, i.e. T will be eventually executed and committed.*

PROOF. Since the client is honest, he eventually sends the contract call to at least t_i honest and reachable service providers in $\mathcal{E}_{C_i}$ (for all $1 \leq i \leq k$), who execute the contract call and return the results to the client. Once the client has received enough responses (i.e. a quorum from each contract), he assembles and broadcasts the transaction for the underlying cryptocurrency. Since there are no conflicting transactions, this transaction will eventually be committed to the chain. □

VIII. IMPLEMENTATION

In this section, we describe a Bitcontracts Python library for creating and running smart contracts that can be used with a backend implementation for arbitrary cryptocurrencies supporting the requirements listed in Section III-D. We also describe a Bitcontracts backend for cryptocurrencies compatible with Bitcoin script.

A. Python Library

Our Python library provides a base class from which all smart contract classes must be derived. Once deployed, the smart contract is an object that is stored serialized on the blockchain. When a contract is run on a service provider, the library (after checking the code and state hashes) first re-instantiates the contract object based on the state provided by the client and then calls the method specified by the client on this object with the provided inputs. Once the method terminates, the library creates a list of state changes from the previous state to the new state, serializes them, and stores them in a transaction, which the service provider then signs.

The library also provides an API to smart contracts. In our prototype, this API is limited to basic functions, such as getting the smart contract balance, creating transfers of funds, or calling other smart contracts, as well as decorators that allow declaring functions as private (i.e. only callable by other functions of this contract) or public (i.e. callable by anyone). An example contract as well as a step-by-step description of its execution is shown in Appendix B. Other API functionality, e.g. some primitives such as getting the caller identity can easily be added. Other functions supported in Ethereum, such as retrieving the current block hash or the identity of the miner would require support from the underlying cryptocurrency, and cannot be added for Bitcoin.

For the execution of the smart contract, a separate execution environment is set up. In our prototype, this is currently a simple subprocess. However, in a production environment, contract execution needs to be executed in a sandboxed environment, e.g. by running the code in a Docker container, since the contract code is not trusted by the service provider.

To ensure that all contract calls can achieve a quorum of service providers, steps should be taken to ensure deterministic execution of the contract code, e.g. by controlling the randomness source available to the sandbox and by disallowing multi-threading.

B. Instantiation for Bitcoin-like Currencies

Transactions in UTXO-based cryptocurrencies consist of multiple inputs and multiple outputs (that can later again be inputs to a transaction). A chain of three Transactions resulting from contract deployment and different calls is shown in Figure 3. Transactions resulting from a contract execution using Bitcontracts have the components described in the following.

Contract Input. The contract input is an output from the previous contract call. We describe it in more detail below. A contract creation transaction does not have any contract inputs.

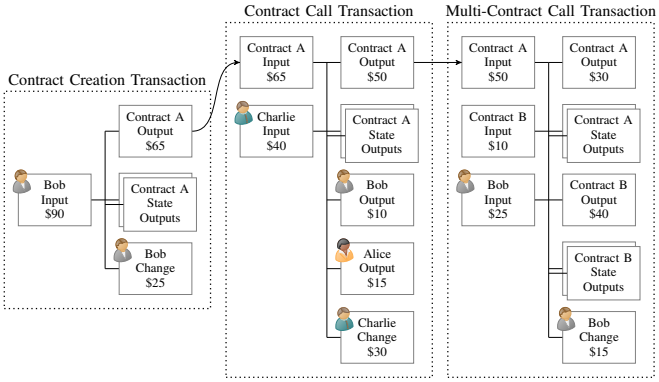

Fig. 3: **Contract Transactions in UTXO-based cryptocurrencies.** In the transaction on the left, Bob creates a smart contract and funds it with $65, which results in a transaction containing a contract output with the funds stored in the contract, several state outputs for persistent storage and an output returning the change to Bob. In the second transaction, Charlie calls the smart contract and sends some funds to it. The contract execution causes the contract to pay out money to Alice and Bob, i.e. in addition to the contract and state outputs, the transactions contains outputs for Alice and Bob, as well as a change output for Charlie. In the transaction on the right, Bob called contract A, which then called contract B. The contract and state outputs of contract A are listed first, then the same for contract B, and the change output to Bob is listed last.

Client Inputs. Any Bitcontracts transaction can have zero or more client inputs. These inputs are used to send funds to the contract.

Contract output. This output holds the balance of the smart contract and is locked by a Bitcoin script specifying a multisig condition. We use a P2SH output with a redeem script containing an m-out-of-n-multisig condition. The parameters m and n as well as the public keys included in it, are chosen by the creator of the smart contract and maintained by the providers throughout calls to it. The rules for standard transactions of Bitcoin and related cryptocurrencies allow for $n \leq 15$ in such redeem scripts (which effectively limits the maximum size of the execution set to $|\mathcal{E}| = 15$). The redeem script also contains the hash of the code and the hash of the current state of the smart contract. These values are pushed to the stack and dropped, thus no additional efforts are required to redeem the balance output. They must still be included, s.t. the provider can check the code and state of the smart contract, received alongside the previous transaction against the hashes contained in it.

State Outputs. These outputs hold the state changes of the contract call, i.e. all variables in the state that were altered during this execution. State changes are stored as a key-value-mapping from variable names to their new values. Using Bitcoin's `OP_RETURN` opcode, up to 80 bytes can be stored in an output that is marked as non-redeemable, i.e. not stored in the UTXO set of a Bitcoin client. However, due to Bitcoins transaction propagation rules only one `OP_RETURN` output per transaction is allowed which is rather limiting.

Several workarounds to this limitation are known and were discussed by Sward et al. [42]. Our implementation uses multisig outputs with three fake public keys containing our data. Bitcoin allows storage of up to three public keys (65 bytes each) in standard multisig outputs (i.e. non-P2SH), which allows storing 195 bytes with an overhead of 15 bytes per output. By arranging the state outputs contiguously, data recovery is straight forward and no additional overhead is incurred do to specifying data locations. With a maximum transaction size of 100KB in Bitcoin we can store up to 92KB of state updates. Note that, while this limits the number of named contract fields that exist at the initialization of the contract, the state size itself is not limited, e.g. a contract using a list or a dictionary can grow to arbitrary sizes.

Client Outputs. These outputs pay money to clients. They can be payouts from the smart contract, or change outputs for a client using an input larger than the value he intended to send to the contract.

To create a smart contract, the client uses one or multiple of his UTXOs as inputs to a transaction that has a contract output with the initial contract funds, state outputs containing the initial state and client outputs, e.g. a change output for the client. In Figure 3 on the left, we show an example transaction, in which Bob creates and funds a Bitcontracts contract.

To call a smart contract, the client first has to assemble the contract state. He does that by iterating through the contract transactions and applying the state updates from each of them. Note, that this can even be done using a lightweight client, i.e. the client does not need to download the full chain of the underlying cryptocurrency. Once the state is assembled, he calls the smart contract by contacting the service providers as described in Section IV-C and shown in Figure 2. The service providers perform the required checks (e.g. matching contract and state hashes), execute the contract and then assemble a transaction. The transaction again contains a contract output as described above, state outputs containing the state changes and potentially client outputs. An example is shown in Figure 3 in the middle, where Charlie sends some funds to a smart contract, which causes a payout to Alice and Bob and the return of some change to Charlie.

A smart contract call may include subcalls to other smart contracts. In such a case, the client provides all required information for all involved smart contracts to the service provider and contacts the necessary service providers from all executing sets (cf. Section IV-D). The service providers execute the contract as described above and when assembling the transaction ensure that the contract and state outputs are ordered in the order in which the contracts appeared in the call chain. For example, in the last transaction in Figure 3 contract B was called by contract A, therefore the outputs for contract A are listed first.

IX. Evaluation

In this section, we first evaluate Bitcontracts that is run on top of popular legacy blockchain platforms (Section IX-A). After that, we evaluate storage and cost of popular real-world smart contracts on top of Bitcontracts (Section IX-B).

A. Bitcontracts on Legacy Cryptocurrencies

For our first evaluation, we consider running Bitcontracts on top of 6 popular legacy cryptocurrencies (Bitcoin, Bitcoin

TABLE II: **Bitcontracts cost evaluation.** The table shows 1) transaction fees in popular cryptocurrencies and 2) the cost of Bitcontracts transactions per state change for each currency based on transaction fee data from 2020-06-08. The table also shows 3) maximum transaction size for each currency and 4) the maximum state change per transactions for Bitcontracts on each currency, as well as 5) the maximum state-change throughput. In 6) these costs and limits are compared to the current Ethereum system (without Bitcontracts).

		Bitcontracts						
		BTC	BCH	LTC	DASH	DOGE	ZEC	6) Ethereum
Storage Cost	1) Transaction fee ($/KB)	0.73	0.005	0.060	0.008	0.000	0.068	-
	2) Bitcontracts state-change cost ($/KB)	0.80	0.006	0.065	0.009	0.000	0.073	1.44 - 5.75
Max. Storage	3) Maximum tx size	100KB	100KB	100KB	100KB	100KB	2MB	-
	4) Bitcontracts max. state-change per tx	92KB	92KB	92KB	92KB	92KB	1.86MB	16 - 63KB
Throughput	5) Bitcontracts max. Throughput (KB/s)	1.5	49.0	6.1	12.2	15.3	12.2	1.0 - 4.2

Cash, Litecoin, Dash, Dogecoin, Zcash) and we compare the costs for data changes as well as the throughput in terms of the amount of data changed to Ethereum. Ethereum follows a roughly similar resource management model, where transaction fees depend on computation complexity and state change storage. We also discuss separately the costs of smart contract execution in others systems, e.g., EOS, that follow a different fee model.

Scalability. As mentioned above, the multisig functionality of Bitcoin and related cryptocurrencies enable execution sets up to the size of $|\mathcal{E}| = 15$ service providers.

Key management overhead. The key management overhead for service providers is low regarding storage, computation, and communication. Each service provider only needs to store a single private key that he can use for all contracts for which he is responsible. Each service provider needs to only produce one signature for each contract execution and there is no communication required between service providers.

Computation. The main focus of our work is contract calls that are similar computations as ones seen today in Ethereum, i.e. in the order of milliseconds. However, nothing limits the system from running more complex contracts similar to those supported by ACE [50], Arbitrum [25] or Yoda [17]. We do not provide a detailed evaluation of the speed of smart contract execution, since contract calls are simply Python program executions and thus measuring their execution speed would not provide any new insights and the cost for contract execution is mainly dominated by the on-chain storage cost described below.

To illustrate this, we compared the execution cost of quicksort in an Ethereum contract with the cost of executing a comparable Python implementation on an AWS t2.micro instance. To sort 2048 integer elements, the Ethereum implementation requires 6.5 million gas (close to the block gas limit), which costs roughly USD 59 (based on the fee prices of 2020-06-08), while on the t2.micro instance, the Python implementation takes less than 6 milliseconds to execute, which corresponds to an effective total execution cost of less than USD $3 \cdot 10^{-7}$, if we assume that this is executed on every service provider in an executing set of maximum size 15[4].

[4]Of course, this does not imply that fees will be this low in practice, since service providers will need to be profitable and therefore fees in practice will be determined by a market and are hard to predict, but it shows that execution in Bitcontracts is generally cheap.

Communication. The communication cost is low, since every call only requires a query to each involved service provider (max $|\mathcal{E}| = 15$) and its response. The latency until a client receives the response from the service provider consists of one Internet round-trip-time plus the time required for the data transfer, which depends on the size of the state changes. For most contracts this would be in the order of milliseconds or a few seconds at most for contracts with many state changes. We do not evaluate this experimentally, as Internet communication latency has already been studied extensively elsewhere. The latency for transaction confirmations in blocks is the same as the latency for other transactions in the underlying blockchain (e.g. on average 10 minutes in Bitcoin). Note that, as is the case with other transactions, Bitcontracts contract calls can also be executed based on unconfirmed transactions, i.e. multiple Bitcontracts transactions for the same contract can be included in the same block.

Storage cost. The first interesting evaluation metric of Bitcontracts is its storage cost. In Table II we first show for reference the current transaction fees (next block inclusion as of 2020-06-08) in popular blockchain platforms. After that, we show the storage cost of Bitcontracts per KB of state changes for the same blockchain platforms. As can be seen from the table, the storage cost overhead caused by Bitcontracts is very small.

We also compare the storage cost of Bitcontracts to Ethereum. As can be seen from Table II, an Ethereum contract storing 1KB of data would cost between $1.44 (if no values are changed from zero to non-zero) and $5.75 (if all changed values are from zero to non-zero), which does not include any computation. Compared to this, the on-chain storage fees of Bitcontracts are significantly smaller ($0.80 or less).

Maximum storage. The second relevant evaluation metric is the maximum amount of data (i.e., state changes) that can be stored per transaction. Table II shows that in Bitcoin, Bitcoin Cash, Litecoin, Dash, and Dogecoin one can store 195 bytes per 210 byte output, and thus the maximum storage per transaction is limited to at most 92KB, due to their standardness rules that will not propagate transactions larger than 100KB. In Zcash, the maximum transaction size is only limited by the maximum block size (2MB), which allows storing data of up to 1.86MB per transaction. We note that these limits do not restrict the overall size of the state of a contract, but only the number of state changes per contract call. In addition,

these limits are affected by other parts of the transaction. For example, if a transaction has a lot of client inputs or outputs, the limit for data storage is reduced accordingly.

In contrast, as can be seen from from Table II, a current Ethereum contract can only change between 16 and 63KB of storage (depending on whether values are set from zero to non-zero) in one transaction given the current block gas limit. Such a transaction would completely fill a block. Using Bitcontracts on Ethereum would allow increasing this limit. Since Bitcontracts does not require the state transitions to be stored in storage and only requires them to be visible in a transaction, they could simply be sent as transaction data, which requires less gas and thus would theoretically allow to store up to 625KB of data per transaction. In practice this would be slightly less, depending on the quorum size and the resulting signature verification cost.

Throughput. Another meaningful evaluation metric for Bitcontracts is throughput. We evaluate throughput by measuring the amount of state-update data the system can process per second. This throughput value depends on the block interval and the maximum storage per block of the underlying blockchain platform. As shown in Table II, executing smart contract using Bitcontracts on top of legacy platforms that do not support contracts natively compares favorably to Ethereum's throughput in terms of the possible amount of changed data.

Costs in different fee models. Some cryptocurrencies, such as EOS, follow a different fee model. Instead of paying for computation and state changes directly, participants in EOS stake funds, i.e., they lock them for some amount of time, and in return they are allowed to use a fraction of the computational and bandwidth resources in proportion to their staked funds. Thus, the main cost for transactions comes from the opportunity cost of not using the staked funds. Storage (called "RAM" in EOS) is bought (and can be traded), but is different from Ethereum and Bitcontracts as participants do not pay for state changes, but instead pay for owning RAM. The cost for 1KB of storage in EOS is $0.15 (as of 2020-06-08) which is more expensive than newly allocating storage in Bitcontracts (except on top of Bitcoin), but changing already allocated storage is free of charge.

B. Popular Ethereum Contracts on Bitcontracts

In our second part of the evaluation, we analyze storage requirements and transaction costs for popular real-world smart contracts, if they were executed in Bitcontracts. We obtained our evaluation data set by crawling the Ethereum blockchain for several weeks (in October/November 2020) and collected smart contract execution data from 130k blocks, from which we extracted transaction information for all transactions of the most popular 100 contracts (based on transaction count). This resulted in a data set containing 10 million contract call transactions. For each transaction, we collected the number and the size of the state changes, the number of involved contracts, the number of clients receiving funds and the Ethereum gas cost.

We then use this information to calculate the transaction size resulting from a potential similar contract execution

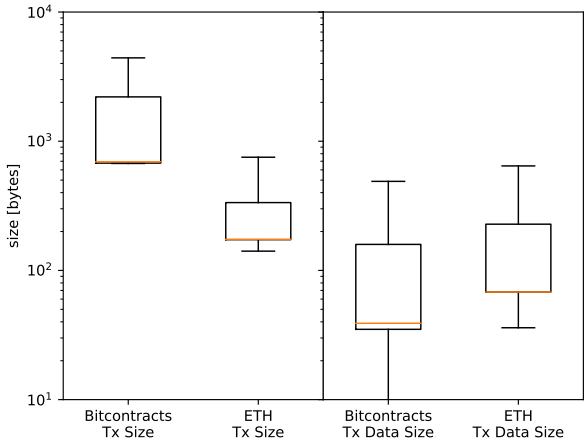

Fig. 4: Full transaction size (left) and transaction data size (right) comparison between Bitcontracts and Ethereum. In Ethereum, the transaction data size includes all transaction inputs, such as function arguments. In Bitcontracts, the transaction data size is the size of all state outputs of the transaction. In the box plots, the whiskers show the 2nd and 98th percentile respectively. The orange line (coinciding with the bottom of the boxes in all but the third box) shows the median value and the bottom and top of the boxes show the 25th and 75th percentile, respectively.

in Bitcontracts and, based on this size, the resulting costs on different chains, such as Bitcoin. The size of a contract transaction in Bitcontracts depends on the number of involved contracts, the sizes of their executing sets, the number of clients receiving funds from the contract and the number and size of the state outputs (cf. Section VIII-B), which is derived from the number of state changes and the size of the storage that is changed. This allows a straight forward calculation for the size and cost of the resulting transaction.

Transaction Size. Figure 4 shows a comparison of the resulting transaction sizes when executing the contract calls in Bitcontracts with executing sets of size 5 and a quorum size of 3. The transaction size distribution for transactions in Bitcontracts is shown on the left next to the size distribution for the same transactions in Ethereum.

The median transaction size in Bitcontracts is 693 bytes, which is only roughly 3 times as large as a basic Bitcoin transaction with one input and two outputs (226 bytes). Transactions are generally larger in Bitcontracts than in Ethereum (which has a median transaction size of 174 bytes), which was expected, since the quorum based off-chain execution requires additional signatures in the transactions and adds some additional overhead for each involved contract, namely the contract inputs and outputs. The right side of Figure 4 compares the transaction data size between Bitcontracts and Ethereum. In Bitcontracts, the transaction data size includes state change outputs. In Ethereum, the transaction data size consists of the transaction inputs, such as function arguments. Our analysis shows that the base cost (that is largely independent of the executed contract) accounts for most of the transaction size and that storing state changes on chain is not the main contributor to the size difference. In fact, transaction data is smaller in most transactions in Bitcontracts compared to Ethereum, with median data sizes

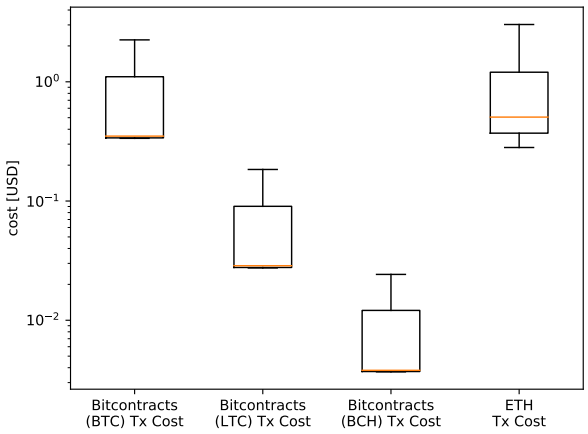

Fig. 5: Transaction cost comparison between Bitcontracts and Ethereumas box plots with the whiskers showing the 2nd and 98th percentile respectively. The orange line (coinciding with the bottom of the boxes in all but the last box) shows the median value and the bottom and top of the boxes show the 25th and 75th percentile, respectively.

of 39 bytes and 68 bytes, respectively, since inputs such as function arguments do not need to be stored in Bitcontracts. This indicates that many contracts change only a small amount of storage data compared to the size of their input parameters.

Throughput. Considering only contract executions (i.e. excluding pure money transfers), based on the collected transaction data, Bitcontracts could support a throughput of 1.8 transactions per second (tps) if run on top of Bitcoin, 7.0 tps on Litecoin, and 9.6 tps on top of Bitcoin Cash (the first two with, the latter without SegWit). Ethereum, for the same set of transaction, supports a throughput of 10.2 tps, which shows that Bitcontracts can achieve a throughput for contract calls on top of legacy cryptocurrencies (Litecoin and Bitcoin Cash) in the same order of magnitude as the throughput achieved by a platform purposefully built to support smart contracts. The throughput of Bitcontracts on top of Bitcoin is lower, but this is expected, since Bitcoin's throughput is more limited even for normal transactions with an average of 3.7 tps.

Transaction cost. Figure 5 shows the transaction costs (in USD) for transactions in our collected set of contract calls for the 100 most popular contracts. The plot compares the costs of these transactions in Bitcontracts on top of Bitcoin, Litecoin, and Bitcoin Cash with the costs of the same transactions in Ethereum (based on transaction fee data from 2020-06-08). We see that the transaction cost distribution for Bitcontracts on Bitcoin (median cost $0.35) has a similar range as the cost distribution of the same transactions in Ethereum (median cost $0.51), while executing them on top of other legacy cryptocurrencies, such as Litecoin (LTC) or Bitcoin Cash (BCH), is much cheaper (median is $0.03 and $0.004, respectively). Note, that this is the on-chain transaction cost, i.e. service provider fees are not included. However, as discussed in Section IX-A, computation time (and thus execution fees) is relatively cheap and the on-chain transaction fees are likely to be the dominant cost factor.

X. Conclusion

In this paper, we introduced a new system, Bitcontracts, that extends legacy blockchains such as Bitcoin with Ethereum-style smart contracts without changes to the base protocol. Bitcontracts achieves this by executing smart contracts in service providers with a quorum based trust model and leveraging the consensus protocol of the underlying blockchain. Bitcontracts only requires the underlying blockchain to provide four basic properties—that are supported by most popular blockchain systems. Our implementation and evaluation show that running smart contracts on top of legacy blockchains is feasible and cost-effective in practice.

References

[1] "CoinMarketCap," 2020, https://coinmarketcap.com/.
[2] "Ripple," 2020, https://ripple.com/.
[3] "Stellar," 2020, https://www.stellar.org/.
[4] K. M. Alonso and KOE, "Zero to monero: Multisig chapter," https://github.com/UkoeHB/Monero-RCT-report/blob/master/multisig_chapter-1-0.pdf, 2018.
[5] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *EuroSys Conference*, 2018.
[6] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on bitcoin," in *IEEE Symposium on Security and Privacy (SP)*, 2014.
[7] A. Back, "Hashcash-a denial of service counter-measure," 2002, http://www.hashcash.org/hashcash.pdf.
[8] W. Banasik, S. Dziembowski, and D. Malinowski, "Efficient zero-knowledge contingent payments in cryptocurrencies without scripts," in *European Symposium on Research in Computer Security (ESORICS)*, 2016.
[9] I. Bentov and R. Kumaresan, "How to use bitcoin to design fair protocols," in *Annual Cryptology Conference (CRYPTO)*, 2014.
[10] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International Conference on the Theory and Application of Cryptology*, 2001.
[11] F. Brasser, U. Muller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: Sgx cache attacks are practical," in *USENIX Workshop on Offensive Technologies (WOOT'17)*, 2017.
[12] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo, "Zero-knowledge contingent payments revisited: Attacks and payments for services," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.
[13] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.
[14] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.
[15] R. Cleve, "Limits on the security of coin flips when half the processors are faulty," in *ACM symposium on Theory of computing*, 1986.
[16] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "Fastkitten: Practical smart contracts on bitcoin," in *USENIX Security Symposium*, 2019.
[17] S. Das, V. J. Ribeiro, and A. Anand, "Yoda: Enabling computationally intensive contracts on blockchains with byzantine and selfish nodes," in *Network and Distributed System Security Symposium (NDSS)*, 2019.
[18] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*, 2015.
[19] J. Dilley, A. Poelstra, J. Wilkins, M. Piekarska, B. Gorlick, and M. Friedenbach, "Strong federations: An interoperable blockchain solution to centralized third-party risks," *arXiv preprint arXiv:1612.05491*, 2016.
[20] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual International Cryptology Conference (CRYPT)*, 1992.

[21] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *IEEE Symposium on Security and Privacy (SP)*, 2019.
[22] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.
[23] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *EuroSys*, 2010.
[24] A. Juels, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," in *Networks and Distributed System Security Symposium (NDSS)*, 1999.
[25] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *USENIX Security Symposium*, 2018.
[26] A. Kiayias, H.-S. Zhou, and V. Zikas, "Fair and robust multi-party computation using a global transaction ledger," in *Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.
[27] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *IEEE symposium on security and privacy (SP)*, 2016.
[28] R. Kumaresan, T. Moran, and I. Bentov, "How to use bitcoin to play decentralized poker," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.
[29] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing," in *USENIX Security Symposium*, 2017.
[30] S. D. Lerner, "Rsk white paper overview," 2015, https://docs.rsk.co/RSK_White_Paper-Overview.pdf.
[31] J. Lyles, "A monero multisig user's guide," https://blog.keys.casa/a-monero-multisig-users-guide/, 2019.
[32] P. McCorry, C. Buckland, S. Bakshi, K. Wüst, and A. Miller, "You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies," in *Workshop on Trusted Smart Contracts*, 2019.
[33] A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *Financial Cryptography and Data Security (FC)*, 2019.
[34] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.
[35] P. Moreno-Sanchez, T. Ruffing, and A. Kate, "Pathshuffle: Credit mixing and anonymous payments for ripple," *Privacy Enhancing Technologies (PETS)*, no. 3, 2017.
[36] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, https://bitcoin.org/bitcoin.pdf.
[37] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016, https://lightning.network/lightning-network-paper.pdf.
[38] I. Ray and I. Ray, "Fair exchange in e-commerce," *ACM SIGecom Exchanges*, vol. 3, no. 2, 2002.
[39] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.
[40] SerHack, "Mastering monero," 2018, https://masteringmonero.com/book/Mastering%20Monero%20First%20Edition%20by%20SerHack%20and%20Monero%20Community.pdf.
[41] A. Singh, K. Click, R. M. Parizi, Q. Zhang, A. Dehghantanha, and K.-K. R. Choo, "Sidechain technologies in blockchain networks: An examination and state-of-the-art review," *Journal of Network and Computer Applications*, vol. 149, p. 102471, 2020.
[42] A. Sward, I. Vecna, and F. Stonedahl, "Data insertion in bitcoin's blockchain," *Ledger*, vol. 3, 2018.
[43] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.
[44] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium*, 2018.
[45] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.
[46] Y. Wang and Q. M. Malluhi, "The limit of blockchains: Infeasibility of a smart obama-trump contract," *Commun. ACM*, vol. 62, no. 5, 2019.
[47] B. Wiki, "Zero Knowledge Contingent Payment," https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment.
[48] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.
[49] K. Wüst and A. Gervais, "Do you need a blockchain?" in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2018.
[50] K. Wüst, S. Matetic, S. Egli, K. Kostiainen, and S. Capkun, "ACE: Asynchronous and Concurrent Execution of Complex Smart Contracts," in *ACM Conference on Computer and Communications Security (CCS)*, 2020.

APPENDIX A
BACKGROUND ON SMART CONTRACTS

Smart contracts [43] are decentralized and self-enforcing digital contracts. A typical smart contract enables contract participants to load money or other assets to an account that is controlled by the contract. The contract's code defines the logic and the conditions based on which the contract may then transfer the loaded money or other assets to different parties, such as the contract's participants.

Most blockchains, like Bitcoin, support simple *scripts* that are primarily used to authorize payments. In this paper, we do not consider such scripts expressive smart contracts. Few blockchains, like Ethereum, provide built-in support for Turing-complete programming languages, and thus, in principle, enable developers to write arbitrary contracts, but in practice Ethereum contracts are constrained by gas limits, which are needed to keep the consensus process efficient.

Ethereum's trust model allows users to choose which particular contracts they decide to trust. If a user participates in a smart contract (e.g., by loading funds to it), he *implicitly* trusts and agrees with the specification of that contract, which is defined by the contract's code. Such trust decisions are contract-specific, as the same user does not need trust other contracts in the same system and is not affected by their execution results.

Ethereum-style smart contracts *cannot* implement all possible contracts, as is discussed in more detail in [46]. One example is contracts that require fairness for revealing input values (e.g., Alice learns Bob's secret if and only if Bob learns Alice' secret). Another limitation of purely digital smart contracts (even with secret computation) is that they cannot enforce control over physical items [49]. Despite such limitations, Ethereum-style smart contracts are widely seen as very useful enablers for various business applications. Thus, the focus of this paper is on *enabling Ethereum-style contracts on legacy blockchains*.

If contracts with private computation are needed, one option is to *complement* Bitcontracts with secure multiparty computation (MPC) techniques. While classical MPC system cannot guarantee fairness without honest majority [15], blockchain-based solutions that leverage deposits and penalties can alleviate fairness concerns [6], [9], [28], [26]. Such privacy protections can be implemented by the contract developer manually or one can use automated contract compilers like HAWK [27].

The second option is to leverage TEEs, similar to Ekiden [14] and FastKitten [16]. The main benefit of this approach is its ease of implementation and its efficiency, in

contrast to cryptographic primitives used in MPC protocols and systems like HAWK. The drawback is that information leakage even from a single TEE violates privacy.

Appendix B
Example Contract

An example for a simple faucet smart contract is shown in Figure 6. The contract allows anyone to top up the faucet with some funds or withdraw money, if funds are sufficient. For each withdrawal, it also increments a counter that is stored in persistent storage. The code shows how API functions for retrieving the contract balance (Line 19) and for sending funds (Lines 21-25) can be used.

```
1  from btcsc.scexecution.base.CSmartContractBase import (
2      CSmartContractBase, public, private)
3  from ISmartContractUtility
         import ISmartContractUtility as util
4
5  class Faucet(CSmartContractBase):
6
7      def __init__(self):
8          super(Faucet, self).__init__()
9          self.withdrawals = 0
10         self.topups = 0
11
12     @public
13     def fill(self, amount):
14         util.IncreaseBalance(self.current_contract, amount)
15         self.topups += 1
16
17     @public
18     def drain(self, amount, address):
19         balance = util.GetBalance(self.current_contract)
20         assert(balance >= amount)
21         util.TransferFromBalance(
22             self.current_contract,
23             amount,
24             address
25         )
26         self.withdrawals += 1
```

Fig. 6: **A simple faucet smart contract implemented in Python.** The contract allows anyone to top up the faucet with some funds or withdraw money.

Figure 7 shows example transactions in which this contract is deployed and executed in a UTXO-based system. Bob is the contract creator and as such creates the contract creation transaction. First, he chooses a set of service providers and we assume that he chooses $\mathcal{E} = \{P_1, P_2, P_3\}$ with a quorum threshold of $t = 2$. He then creates a 2-out-of-3 multisignature output for the public keys of these three service providers that contains initial contract funds. This output is the *contract output* and it also needs to contain the hash of the contract code as well as the hash of the current state (i.e. the state after running the constructor of the contract class)[5]. The state in this case contains the two state variables `withdrawals` and `topups`, which store the number of withdrawals and top-ups of the faucet, respectively and which both have a value of zero. The actual values of the state variables are stored in separate outputs (in this case one output suffices), using the method described in Section VIII-B. Finally, he creates an output that refunds his change. He then creates and signs a transaction that uses one of his UTXO as input and broadcasts this transaction.

At this point, the contract has been created. Now, potential participants need to be informed about the contract code and its contract output. This can happen privately, through a public website, or the code could be stored in the deployment transaction as well. Charlie now wants to use this faucet contract to receive some money from it. He uses the contract output to assemble the current state and sends it, together with the contract code, the previous transaction Tx_A (in this case the contract creation transaction) as well as the function he wants to execute and it's arguments (i.e. "`Faucet.drain(5, address_charlie)`") to P_1, P_2, and P_3.

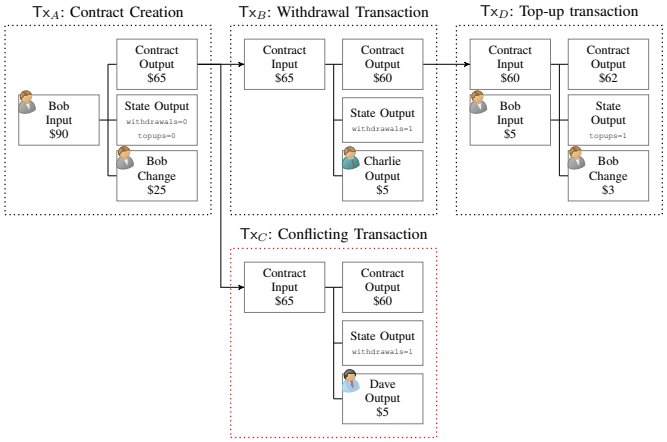

Fig. 7: **Example Contract Execution.** Bob first creates the smart contract, followed by Charlie performing a withdrawal. At the same time, Dave also tries to perform a withdrawal, but due to the state dependency check (linking to the previous transaction output in this case), the transaction cannot be included in the chain. Finally, Bob tops up the faucet with additional funds.

The service providers now hash the received contract code and state and compare the values to the hashes referenced in Tx_A[6]. If the values match, each service provider executes the transaction. The code contains an API call that transfers money, which will cause the library to create a transaction output for the specified address with the specified amount ($5 in this example). The function called by Charlie also updates the `withdrawals` state variable. After the execution of the code is finished, the library compares the new state to the previous state and then creates a state output (or multiple, if necessary) containing only the values that have changed. In this case, `withdrawals = 1`. Each of the service providers create a transaction Tx_B with the mentioned outputs, as well as an updated contract output containing the current balance, contract hash and the new state hash. The previous contract output is used as input of this transaction. They also create a signature on this transaction and send both to Charlie. Charlie then finalizes the transaction by attaching the signatures from two of the service providers to it and broadcasts it to the network.

Assume now that Dave also wants to use this faucet at almost the same time. He performs exactly the same steps as Charlie and will receive a similar transaction Tx_C as result. However, since Tx_B and Tx_C use the same contract output as transaction input, they are conflicting and miners will only accept one of them. In Figure 7, we assume that Tx_B will be

[5]In Bitcoin script, this can be done by using a pay-to-script-hash (P2SH) output, in which the script drops the two hashes before evaluating the multisignatures. The output stores the hash of this script, which ensures that neither the public keys for the multisignature nor the two hashes can be changed.

[6]In Bitcoin script, this is done by recreating the script from the P2SH output and checking if the hash of this script corresponds to the script hash in the output.

included in the chain. If another party now wants to use the faucet, e.g. Bob wants to top up the funds of the faucet, they can do so based on the state given by Tx_B (even before Tx_B is included in a block). In the example, Bob performs such a top-up, which results in a change in the `topups` state variable and thus only this changed variable is stored in the state output.