Bitcontracts: Adding Expressive Smart Contracts to Legacy Cryptocurrencies

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

***Abstract*—In contrast to traditional contracts, cryptocurrency-based smart contracts can provide improved business automation and more transparency. However, not all cryptocurrencies support expressive contracts. For example, Bitcoin only supports a restricted scripting language that is not expressive enough to realize many contracts. Ethereum supports a Turing-complete programming language, but the types of contracts that can be implemented are still severely constrained due to gas limits. Recent research has explored ways to add contract support to legacy currencies like Bitcoin or enable more complex contracts on systems like Ethereum, but such previous solutions have significant security and functional limitations.**

In this paper we propose Bitcontracts, a novel solution to enable generic and expressive smart contracts on legacy cryptocurrencies. The starting point of our solution is a common 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 challenge of this paper is how to realize such a trust model securely and efficiently without modifying the underlying blockchain. Bitcontracts achieves this using two main techniques. First, the state of each contract is stored on the chain which avoids the need to run expensive consensus protocols between the service providers. Second, the validity of each execution result is bound to the latest state of the chain to prevent double-spending attacks. Bitcontracts can be used to retrofit contracts to currencies like Bitcoin or to extend the contract execution capabilities of systems like Ethereum. We also identify a set of generic properties that a blockchain system must support so that expressive smart contracts can be added safely and efficiently, and analyze existing blockchains based on these criteria.

I. Introduction

Smart contracts, popularized by systems like Ethereum, allow nearly arbitrary (Turing-complete) 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 smart contract enables the contract participants to load money to an address or account that is controlled by the contract. The contract's code defines the logic and conditions based on which the loaded money can be moved out of the contract, typically to one of the contract's participants.

Currently, the industry envisions the usage of smart contracts in many applications beyond exchange of cryptocurrencies, IOU payments, or financial settlements. Commonly suggested use cases include financial services, identity management, insurance, and supply chain governance. Such significant business interest is driven by the expectation that, in contrast to traditional contracts and business applications, smart contracts can offer better business process automation, stronger guarantees for the contract parties, and more transparency.

Adding contracts to currencies. While the concept of smart contracts has shown great promise, many currently popular cryptocurrencies, such as Bitcoin, 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, in many cases it is 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 is based on a Turing-complete programming langue, but the types of computations that can be implemented as contracts are highly restricted, due to the built-in gas limits that are needed for the consensus process.

An additional limitation of currently popular blockchains is that they typically support dedicated, niche programming languages. For example, Ethereum contracts must be written in languages like Solidity that can be compiled to EVM bytecode. Most programmers are more familiar with general-purpose programming languages like Python and Java which also have better tools, documentation, and have runtime environments that have been long optimized over decades.

In this paper, our main goal is to design a solution that adds expressive smart contract execution capabilities to existing blockchain systems. The primary usage of our solution is to enhance systems like Bitcoin that have no built-in smart contract support. 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. We also want to enable contract developers to write contracts in their favorite programming language.

Previous work. Recent research has explored different ways to add contract execution capabilities to existing blockchains. However, all such solutions suffer from significant security, deployment, and functional limitations.

For instance, Arbitrum [20] proposes an *off-chain* execution model, where the contract issuers appoint a set of managers who are responsible for executing the contract code off-chain and then communicating the contract execution results back to the chain, where they are included if they are signed by all managers. The main drawback of such solutions is that they require changes to the consensus protocol, and therefore such systems cannot be deployed to legacy systems without modifying the underlying blockchain platform.

Other proposals, such as FastKitten [14], rely on *enclaved execution* and *collaterals*. In FastKitten, contracts are executed inside a single SGX enclave and each contract participant, as well as the enclave's operator, must place a deposit during the contract's initialization to prevent misbehavior. While such solutions are compatible with unmodified legacy currencies, they only support short-lived contracts that are restricted to known participants. In many practical smart contract use cases, the set of contract participants is not known at the time of the contract's deployment. In addition, such a system cannot tolerate enclave compromise. The recently discovered SGX side channels [9], [29], [32], [36], [24] and micro-architectural attacks [11], [10] have shown that TEE compromise is a relevant threat that should be considered. We discuss the limitations of previous solutions in more detail in Section III-B.

Our solution. In this paper, we propose a novel system called Bitcontracts that adds expressive smart contract execution capabilities to legacy cryptocurrencies 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. 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 such a set is acceptable. 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*, where execution results are acceptable 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 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 simplifies deployment and they do not have to run expensive consensus protocols to agree on the current state of the contract which makes our solution more 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 double-spending attacks where the adversary obtains two acceptable quorums for two different execution results affecting the same coins.

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

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

We implemented a prototype of Bitcontracts on top of Bitcoin that supports contracts written in Python. We also evaluated the costs of executing contracts on popular legacy blockchains and show that in most cases the involved transaction fees are small (e.g., few USD cents per contract call).

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

- *New solution:* We propose Bitcontracts, a novel system that adds smart contract execution capabilities to legacy cryptocurrencies in a secure and efficient way.
- *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.
- *Implementation and evaluation:* We implemented Bitcontracts such that it runs on Bitcoin and supports Python contracts, and evaluated contract execution costs across many popular blockchain platforms.

II. Background

In this section, we provide background on blockchain-based cryptocurrencies and smart contracts.

A. Cryptocurrencies

Bitcoin [30] introduced the concept of blockchains to solve the double-spending problem for digital currencies in a fully-decentralized way. Bitcoin uses the so-called *UTXO model*, where funds are bound to transaction outputs that are created either as new coins through mining or as a result of previous transactions. Transactions consume inputs (one or multiple unspent transaction outputs, or UTXO) and create new outputs that sum up to the same value as its inputs. Each output is associated with a script that specifies the conditions to spend it. For example, an output can specify a hash of a public key and its spending is authorized when the transaction containing it is signed by a key that corresponds to the hash. Other conditions, such as requiring signatures by multiple listed keys (*multi-signature* transaction) can be specified as well. Several other blockchain currencies, such as Litecoin, ZCash, Dash, and Cardano, have adopted the same UTXO model.

The second common approach to express transactions in blockchain-based cryptocurrencies is the *account model*. Ripple, Stellar and Ethereum are examples of blockchain systems that adopt this approach. In such blockchains, funds are stored in accounts that are controlled by keys. A transaction does not specify inputs and outputs, but instead specifies a

source and a target account as well as the transaction amount. The transaction then transfers the specified amount from the source to the target account. While the basic model associates one key with every account, such cryptocurrencies can also support other types of spending authorizations, such as shared accounts that require a multi-signature to approve a payment.

Most permissionless blockchain systems use a P2P network for message dissemination. The network is assumed to be well connected and free of eclipse attacks [19], [38]. Other trust assumptions depend on the consensus scheme. For example, in Proof-of-Work (PoW) cryptocurrencies, an honest majority of the mining power is assumed. Temporary forks can happen, and thus the most recent transaction history may change and only eventual consistency is guaranteed.

B. Smart Contracts

Smart contracts [35] 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.

Smart contracts can offer multiple advantages over traditional contracts and business applications. One advantage is improved availability and integrity, because contract execution cannot be unilaterally blocked or changed by any contract party. Another is increased transparency, as the contract's code is publicly recorded on the chain and its execution results can be verified by anyone.

Most blockchain-based currencies 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. However, in practice the types of computations that can be realized as Ethereum contracts are constrained by gas limits, which are needed to keep the consensus process efficient because all miners need to execute all contracts sequentially.

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.

III. Problem Statement

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

A. Motivation

During the last few years, blockchain technology has gathered significant business interest that is largely focused on smart contracts and their applications. 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 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. At the time of writing (June 2019), 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 creating new blockchain platforms 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 that support contracts 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 and time-consuming computations as smart contracts that what is allowed by Ethereum currently.

And finally, the existing smart contract platforms are based on dedicated (often niche) programming languages such as Solidity for Ethereum. Most developers are more familiar with general-purpose languages 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.

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.

B. Limitations of Previous Solutions

Next, we outline the main limitations of previous solutions.

Side-chain execution. One known approach to extend legacy currencies with contracts is to use a *side-chain*. For example, Rootstock (RSK) [25] 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 on RBTC currency. The main limitation of such solutions is that they require users to migrate their funds to a separate currency, although one that is pegged to the original currency.

Off-chain execution. Another approach is to run contract code *off-chain* in few chosen execution nodes. Arbitrum [20], ACE [40], and Yoda [15] follow this approach. In Arbitrum, the contract issuer appoints a set of managers who are respon-

sible 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). In ACE, contracts are executed by sets of service providers, of which a quorum is required to commit results. Because of this, ACE does not require dispute resolution and its trust model is closest to that of Bitcontracts. 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 they require changes to miners and thus such solutions cannot be deployed to legacy blockchains without modifications.

Enclaved execution. The next known approach is to outsource contract execution into a *trusted execution environments* (TEEs) like SGX enclaves. Ekiden [12] 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 loaded funds. Recent research on SGX side channels [9], [29], [32], [36], [24] and micro-architectural attacks [11], [10] 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 [13] 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]. Soon after, this was extended to n-party lotteries [7], playing poker [23] and other MPC protocols that leverage distributed ledgers [21]. In all of these 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 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 [23].

Enclaved multiparty computation. A recent work called FastKitten [14] combines techniques from Ekiden [12] and blockchain-based MPC [6], [7], [23], [21] 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 to 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 (e.g., because one of the users 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). While our goal is to enable more expressive smart contracts, FastKitten actually enables more restricted contracts. 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.

IV. Bitcontracts Overview

In this section, we provide an overview of our solution Bitcontracts. We start by explaining what type of contracts we want to support. After that, we describe our execution model and discuss the challenges of realizing it. Finally, we explain the main ideas of Bitcontracts and define common properties that a blockchain must provide to support it.

A. On Contract Type

The notion of "smart contracts" was introduced more than two decades ago [35] and later popularized by permissionless blockchains like Ethereum. In Ethereum, smart contracts can perform Turing-complete computations (constrained by gas limits) and perform payments using cryptocurrency that is loaded to a contract-controlled account. Such smart contracts can have an unlimited lifespan and arbitrary many participants who do not have to be known from the beginning.

Blockchains in general can be viewed as providing a trusted third party (TTP) with public state. In the case of Bitcoin and similar cryptocurrencies that provide payments that can be extended with some limited conditions (Bitcoin script), this TTP is very restricted, but can e.g. allow for constructions such as payment channels [31], [16]. By adding Ethereum-style smart contracts that allow for Turing-complete computations, the resulting TTP becomes much more powerful, but it is still limited due to not allowing for secret computation and state.

Because of this, Ethereum-style smart contracts *cannot* implement all possible contracts, as is discussed in more detail in [37]. One example is contracts that require fairness for revealing input values, i.e. 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 [39]. Despite such limitations, Ethereum-style smart contracts are widely seen as very useful enablers for various business applications and use cases. Thus, in this paper, *we focus on enabling Ethereum-style contracts on legacy blockchains*.

If contracts with private computation are needed, the first option is to complement Bitcontracts with known secure multiparty computation (MPC) techniques. MPC protocols allow a set of participants to jointly evaluate a function over private inputs. While classical MPC system cannot guarantee fairness without honest majority [13], blockchain-based solutions that leverage deposits and penalize malicious behavior can alleviate such fairness concerns [6], [7], [23], [21]. Such privacy protections can be implemented by the contract developer manually or one can use automated contract compilers like HAWK [22].

The second option to support private contracts is to leverage TEEs, similar to Ekiden [12] and FastKitten [14]. The main benefit of this approach is that it applies to any type of contract without extra developer burden. Enclaved contract execution is also efficient, in contrast to cryptographic primitives used in MPC protocols and systems like HAWK. The drawback is that information leakage even from a single TEEs violates privacy (see Section X-B).

B. 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. Perhaps the simplest such solution would be one where a single service provider that has full access to the funds of a smart contract, on request executes that contract and then performs the resulting payments. However, such a solution requires clients to fully trust this single service provider. While trust concerns could be somewhat addressed using trusted execution environments (TEEs) such as Intel SGX, as used by Ekiden [12] and FastKitten [14], recent research has shown that compromise of TEEs is a very practical threat that needs to be considered. In addition, availability of such a solution would be poor, regardless of TEE compromise.

To improve such a simple solution, an obvious approach is to distribute trust among several service providers, i.e. instead of assuming that one entity behaves honestly, one trusts a set of service providers collectively, as is done in system like Arbitrum [20]. 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*, in which the contract creator can choose the requirements for acceptable execution results per contract. Namely, the creator of a contract chooses a set $\mathcal{E}$ of 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.

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 $|\mathcal{E}|$. 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 $t = 1$.

Our trust model modifies the typical trust assumptions of smart contract systems slightly. For example, in Ethereum, the specification of a smart contract is defined by its code (cf. Section II-B). In our system, the specification also includes a set of service providers and the threshold. Importantly, all users can still decide if they trust and agree with this specification. And 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.

C. Challenges

The primary technical challenge that we solve in this paper is how to realize the above outlined execution and trust model securely and efficiently. Next, we discuss why simple solutions fail to solve the problem.

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.

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 consensus protocol separately. This option also increases transparency towards the clients, as they can individually verify the correctness of every execution result.

At first glance, it may seem that storing the state on the blockchain is enough to ensure consistency between the service providers (and thus the integrity of the smart contracts they execute); this is however 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 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

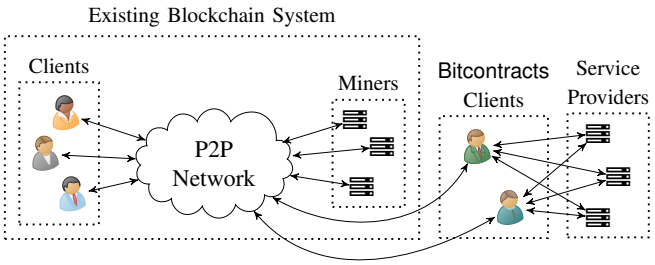

Fig. 1. **Bitcontracts overview.** Bitcontracts extends existing blockchain systems without changing their protocol, i.e. existing nodes such as clients and miners are agnostic to Bitcontracts. Bitcontracts clients interface with the blockchain and Bitcontracts service providers. Service providers are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

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., ones based on PoW).

D. Overview of Bitcontracts

Next, we explain the main ideas of our solution, Bitcontracts that is illustrated in Figure 1. We combine *off-chain execution* of contracts with *on-chain storage* for contract state. This design allows the service providers to be completely stateless, avoids the need to run expensive consensus protocols and provides transparency towards the contract's client.

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 transition that they return to the client. The client combines the received signatures from t service providers that report the same result so that the required multipart authorization is fulfilled and broadcast the complete transaction to the P2P network. The miners accept the state change transaction if it signed by at least t service providers who control that contract's account.

It might seem counterintuitive to have the client assemble and broadcast the final transaction, as he can then choose not to broadcast it if e.g. the results are unfavourable to him. However, this is not an issue, since contract execution is deterministic, i.e. the client can already know the results of the contract execution before initiating the contract call. This is the same as in other smart contract systems such as Ethereum.

To prevent the attacks described in Section IV-C where the adversary obtains two valid quorums for the same contract state, 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. This mechanism in our solution prevents attacks, where the same coins can be spent more than once, even if the blockchain experiences short-lived forks and at the same time allows usage of arbitrary quorums.

Finally, Bitcontracts enables contracts where a single transaction perform 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.

E. Cryptocurrency Properties

Above we informally introduced four properties 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, e.g. a single executing node, is chosen, fewer properties can be sufficient.

Next, we specify these four properties more precisely in the format of interfaces. This allows us to keep our system design (Section V) agnostic of the underlying blockchain platform. Later, in Section VI, 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[1] 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 VI we discuss how this property is provided in account-based systems.

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 VI).

[1]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 [8] signatures.

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.

V. 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 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 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 if he does not intend to make the service available to every entity.

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. 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.

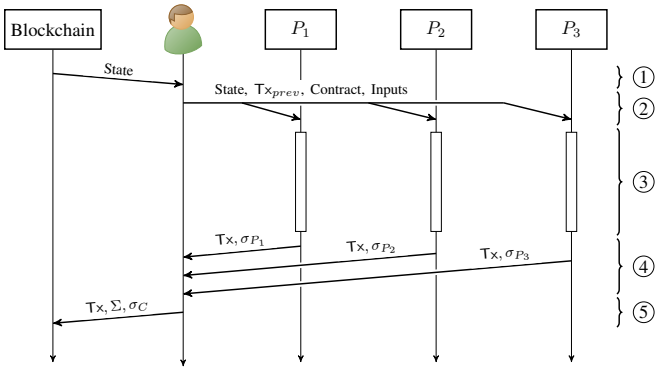

Fig. 2. **Contract call.** To call a smart contract, the client first assembles the state from the blockchain and then sends the state, the previous transaction, and his inputs to the service providers. The service providers then execute the contract call and send the resulting transaction as well as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

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 $\mathcal{E}$ 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.

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.

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).

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 must send 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 V-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.

VI. Property Analysis

In this section we analyze currently popular blockchain systems and explain how they provide the properties that Bitcontracts needs. Table I summarizes our analysis showing that most popular cryptocurrencies provide all required properties, or, in the few exceptions, could be easily extended to do so.

Table I. **Supported Properties.** This Table shows which of the required properties are supported by the most prominent 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	✔	✔	✔	✔

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 [34] 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 [33] 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. 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 multisignatures.

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], [26]. While this is enough to be compatible with our system, it requires that the service providers interact, 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.

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 support atomic transactions with multiple sources and destinations, however this could easily be added, similar to Stellar.

VII. Security Analysis

In this section, we analyze the security, safety, and liveness of Bitcontracts contracts. We show that Bitcontracts guarantees correctness if less than a quorum of an executing set is compromised and liveness if an honest quorum is reachable.

A. Security & Safety

We first consider security and safety for contract calls. We say that a contract execution is correct 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, 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 is executed correctly.

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 control-flow 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. 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 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, which provides a unique serialization.

Since a contract call that involves multiple contracts requires a quorum for each involved contract, the above applies to all contracts independent of whether they involve other contracts or not. □

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.

```python
from btsc.scexecution.base.CSmartContractBase import (
    CSmartContractBase, public, private)
import ISmartContractUtility as util

class Faucet(CSmartContractBase):

    def __init__(self):
        super(Faucet, self).__init__()
        self.withdrawals = 0

    @public
    def fill(self, amount):
        util.IncreaseBalance(self.current_contract, amount)

    @public
    def drain(self, amount, address):
        balance = util.GetBalance(self.current_contract)
        assert(balance >= amount)
        util.TransferFromBalance(
            self.current_contract,
            amount,
            address
        )
        self.withdrawals += 1
```

Fig. 3. **A simple faucet smart contract.**

B. Liveness

Since liveness is clearly not achievable if no honest quorum of service providers is reachable by the client, we add an additional requirement regarding reachability in comparison to safety, and make the following claim:

CLAIM 2. 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 at least t_A uncompromised service providers are reachable by the client, then contract A provides liveness.

PROOF. Here, we need to consider two cases. First, for contract calls that do not involve other contracts, if at least t_A service providers of $\mathcal{E}_A$ are honest and reachable, requests from clients will be completed. If a client does not publish the results, this has no effect on the availability of the contract, since the service providers are stateless, i.e. they can still complete requests from other clients based on the state that is currently committed to the blockchain.

If contract calls involve other contracts, they may not be completed, if the executing set of the other contract does not have an honest quorum. However, this does not affect other calls to contract A, since the providers are stateless and therefore contract A can continue to make progress, i.e. liveness is provided. □

VIII. Implementation and Evaluation

In this section, we first 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 IV-E. Second, we describe a Bitcontracts backend for cryptocurrencies compatible with Bitcoin script. Finally, we evaluate our implementation with regards to required on-chain storage and transaction cost.

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). 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 notions of time, such as block height or block timestamps in Ethereum, there are different possibilities for adding similar primitives to the API that we discuss in Section X-C.

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.

An example for a simple faucet smart contract is shown in Figure 3. 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 20) and for sending funds (Lines 22-26) can be used.

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 4. 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.

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

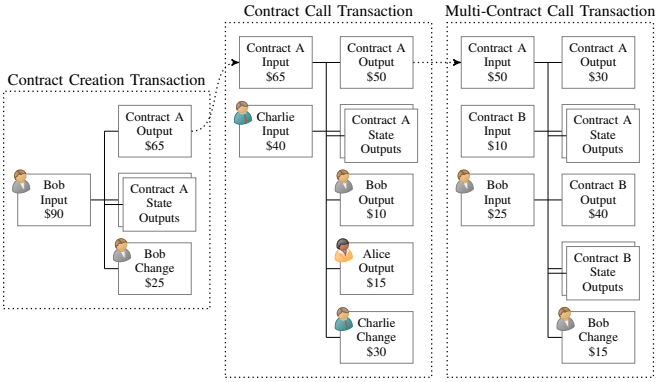

Fig. 4. **Contract Transactions in UTXO-based cryptocurrencies.** In the transaction on the left, Bob creates a smart contract and funds it with $65, which results in a transaction containing a contract output with the funds stored in the contract, several state outputs for persistent storage and an 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.

of Bitcoin and related cryptocurrencies allow for $n \leq 15$ in such redeem scripts. 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. [34]. 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.

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 4 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 V-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 4 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 V-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 4 contract B was called by contract A, therefore the outputs for contract A are listed first.

C. Evaluation

We analyzed the storage requirements and fees for on-chain storage of persistent data. We do not evaluate the speed of smart contract execution since the contracts are simply Python programs. We also do not evaluate the throughput, since it is equal to the throughput of the underlying cryptocurrency in terms of data (except for some overhead as described below) and highly contract dependent in terms of transactions per second. For example, some contracts could result in transactions barely larger than standard Bitcoin transactions, while others might could use up to 100KB, leading to throughputs that are orders of magnitude apart.

The maximum amount of data (i.e. state changes) that can be stored depending on the cryptocurrency is shown in Table II. Since we can store 195 bytes per 210 byte output, the maximum storage is limited to at most 92KB in Bitcoin, Bitcoin Cash, Litecoin, Dash, and Dogecoin, due to their standardness rules that won't propagate transactions larger than 100KB[2]. 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. 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, e.g. if a transaction has a lot of client inputs or outputs, the limit for data storage is reduced accordingly.

In contrast, a current Ethereum contract can only change between 13 and 51KB of storage (depending on whether values are set from zero to non-zero) in one transaction given the current block size limit. However, using Bitcontracts on Ethereum would allow increasing this. 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 much less gas and thus would theoretically allow to store up to 117KB of data per transaction. In practice this would be slightly less, depending on the quorum size and the resulting signature verification cost.

To compare the viability of using Bitcontracts in different cryptocurrencies, we compare the cost for storing state changes in Table II. The fees are given per KB of state changes (i.e. taking into account overhead) based on the fees for next block inclusion as of 2019-06-27. For comparison, current transaction fees are also shown in the table, which demonstrate the low overhead of Bitcontracts. In current Ethereum contracts storing 1KB of data would cost between \$0.95 (if no values are changed from zero to non-zero) and \$3.81 (if all changed values are from zero to non-zero), which does not include any computation. Compared to this, the fees for on-chain storage using Bitcontracts are cheaper in all compatible currencies except for Bitcoin, showing that Bitcontracts is a viable solution in practice.

IX. Further Related Work

We already discuss closely related work and their limitations in Section III-B. Here, we briefly discuss other related work.

Hyperledger Fabric [5] is a permissioned blockchain system that uses a similar execution model to ours in that contracts are executed first, then signed by a set of *endorsers* (similar to service providers) and only ordered afterward. In contrast to our solution, Fabric requires endorsers to be stateful and base the execution on their local state. Because of this, Fabric adds an additional validation phase that checks that the read and write sets of concurrent executions are not in conflict, which is neither required nor possible in our case. In addition, Bitcontracts can extend the functionality of many existing blockchains, including those in the permissionless setting, while Fabric is a full standalone system.

Other related work includes work on state channels [28], [17], [18], [27] that optimistically move on-chain execution of smart contracts to off-chain execution. Contracts are executed by unanimous consent of the contract participants. Such constructions always require a fallback mechanism on-chain and joining such a channel requires an on-chain execution. In contrast, our solution is on-chain regarding the state storage, but off-chain regarding execution, and execution is delegated to service providers that do not need to agree unanimously.

[2]In the case of cryptocurrencies implementing segregated witness, this is defined by a maximum weight. Since the storage in our case is in outputs, this is de facto equivalent to a 100KB transaction size limit.

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. The table also shows 3) maximum transaction size for each currency and 4) the maximum state change per transactions for Bitcontracts on each currency. In 5) these costs and limits are compared to Ethereum.

	Bitcontracts						
	Bitcoin	Bitcoin Cash	Litecoin	Dash	Dogecoin	Zcash	5) Ethereum
1) Transaction fee ($/KB)	14.8	0.01	0.14	0.02	0.00	0.12	-
2) Bitcontracts state-change cost ($/KB)	16.0	0.01	0.15	0.02	0.00	0.13	0.95 - 3.81
3) Maximum transaction size	100KB	100KB	100KB	100KB	100KB	2MB	-
4) Bitcontracts max state-change per tx	92KB	92KB	92KB	92KB	92KB	1.86MB	13 - 51KB

X. Discussion

In this section, we discuss service provider remuneration and hardening, as well as providing a notion of time.

A. Remuneration of Service Providers

To incentivize service providers to participate in Bitcontracts, several remuneration models are possible:

Subscription model. When the service providers are well-established entities, they can provide a subscription model for smart contract execution, similar to existing cloud providers. In this case, they could receive regular payments through arbitrary means, e.g. from the contract creator, to compensate them for their responsibility of executing the smart contract. This could be combined with service level agreements.

Built-in transaction fees. The system can be deployed with fixed (per amount of computation) transaction fees, similar to the gas model of Ethereum, where each operation in the smart contract language is assigned a value based on its computational complexity. For this scenario, similar to Ethereum, the client can specify a "price per computational unit" (the equivalent to gas in Ethereum) which will convert the total computational complexity into an amount in the underlying cryptocurrency, which will get paid to the service providers in the transaction resulting from the execution.

Flexible remuneration. Since smart contracts are written in an expressive language, handling fees can be implemented within the contract itself, i.e., the terms of execution are itself a part of the smart contract. Service providers can inspect these terms and execute the smart contract if the resulting fees are agreeable. The smart contract then initiates payments to the service providers as part of the resulting transaction.

B. Hardening of Service Providers

In order to increase the security of each service provider, they can execute contract code inside TEEs like SGX enclaves. In such deployment, Bitcontracts clients, and all other parties, can verify that the enclave is correctly initialized using remote attestation. The remote attestation evidence, a signed statement by Intel's IAS service, can be recorded on the chain for public verifiability. The attested enclave can create a key pair such that the private key is only known to that enclave and the public key is bound to the attestation evidence so that external parties can then verify signatures made by the enclave.

If TEEs are used on each service provider responsible for a given contract, Bitcontracts provides the following security guarantee. To violate contract execution integrity, the adversary must be able to compromise at least t enclaves. We consider this a difficult task and note that it is a stronger security guarantee than provided by previous solutions like FastKitten [14]. If contract privacy relies on the use of TEEs, then it is sufficient for the adversary to compromise just one enclave to violate privacy (see below). For this reason, we recommend that enclaved execution is complemented with cryptographic protections when th strongest level of privacy is required.

C. Notion of Time

Many smart contracts benefit from having a notion of time, e.g. for commit-reveal schemes that could be used in an auction. Ethereum provides such a notion in two different ways. One of them is a block timestamp, which is added by a miner and does not necessarily show the exact time the block was mined, and the other is the block number. Both of these are coarse-grained, but they still allow clients and a smart contract to share a notion of time.

In Bitcontracts there are different ways of achieving a similar notion. The first assumes that honest service providers are loosely synchronized (i.e. their local clocks are not too far off from the real time), which is a reasonable assumption in practice. In that case, the contract developer (or the API) can use the local clocks of the service providers to provide the time. E.g. if some operation is conditioned on the current time being after some threshold, they can simply use the time directly. For other functions, the client could supply a timestamp and the API on each service provider would check if this timestamp is within some range of the local time.

Another less generic possibility is to rely on primitives provided by the underlying cryptocurrency. For example, Bitcoin-like currencies provide timelocks that could be used for many applications that require a notion of time, e.g. auctions. However, this makes the API blockchain-dependent and is thus less preferable than the first option.

XI. Conclusion

Smart contracts show great promise for various business applications, but unfortunately most cryptocurrencies do not support contracts and those currencies that do support them have serious functional limitations. In this paper, we have proposed a novel system called Bitcontracts that extends existing cryptocurrencies with contract execution capabilities. Our solution enables expressive Ethereum-style smart contracts for unmodified legacy currencies and supports a flexible trust model that can be adjusted to meet the security requirements of various contract deployments.

REFERENCES

[1] CoinMarketCap, 2019. https://coinmarketcap.com/.

[2] Ripple, 2019. https://ripple.com/.

[3] Stellar, 2019. https://www.stellar.org/.

[4] Kurt 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] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

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

[7] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *Annual Cryptology Conference*, pages 421–439. Springer, 2014.

[8] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

[9] Ferdinand Brasser, Urs Muller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *USENIX Workshop on Offensive Technologies (WOOT'17)*, 2017.

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, Baltimore, MD, 2018. USENIX Association.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.

[12] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. 2018.

[13] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369. ACM, 1986.

[14] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. Fastkitten: Practical smart contracts on bitcoin. Cryptology ePrint Archive, Report 2019/154, 2019. https://eprint.iacr.org/2019/154.

[15] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. Yoda: Enabling computationally intensive contracts on blockchains with byzantine and selfish nodes. *arXiv preprint arXiv:1811.03265*, 2018.

[16] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

[17] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment channels over cryptographic currencies. *IACR Cryptology ePrint Archive*, 2017:635, 2017.

[18] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 949–966. ACM, 2018.

[19] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 129–144, 2015.

[20] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 1353–1370. USENIX Association, 2018.

[21] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 705–734. Springer, 2016.

[22] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

[23] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 195–206. ACM, 2015.

[24] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing. In *USENIX Security Symposium*, 2017.

[25] Sergio Demian Lerner. Rsk white paper overview, 2015.

[26] Jacob Lyles. A monero multisig user's guide. https://blog.keys.casa/a-monero-multisig-users-guide/, 2019.

[27] Patrick McCorry, Chris Buckland, Surya Bakshi, Karl Wüst, and Andrew Miller. You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies. In *3rd Workshop on Trusted Smart Contracts*, 2019.

[28] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Christopher Cordi, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *International Conference on Financial Cryptography and Data Security. Springer*, 2019.

[29] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[30] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[31] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[32] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[33] SerHack. Mastering monero. https://masteringmonero.com/book/Mastering%20Monero%20First%20Edition%20by%20SerHack%20and%20Monero%20Community.pdf, 2018.

[34] Andrew Sward, Ivy Vecna, and Forrest Stonedahl. Data insertion in bitcoin's blockchain. *Ledger*, 3, 2018.

[35] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[36] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl 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.

[37] Yongge Wang and Qutaibah M. Malluhi. The limit of blockchains: Infeasibility of a smart obama-trump contract. *Commun. ACM*, 62(5), 2019.

[38] Karl Wüst and Arthur Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.

[39] Karl Wüst and Arthur Gervais. Do you need a blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54, June 2018.

[40] Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. ACE: Asynchronous and Concurrent Execution of Complex Smart Contracts. *IACR Cryptology ePrint Archive*, 2019:835, 2019. https://eprint.iacr.org/2019/835.