FastKitten:
Practical Smart Contracts on Bitcoin

Poulami Das* Lisa Eckey* Tommaso Frassetto§ David Gens§
Kristina Hostáková* Patrick Jauernig§ Sebastian Faust* Ahmad-Reza Sadeghi§
Technische Universität Darmstadt, Germany
* first.last@cs.tu-darmstadt.de
§ first.last@trust.tu-darmstadt.de

Abstract

Smart contracts are envisioned to be one of the killer applications of decentralized cryptocurrencies. They enable self-enforcing payments between users depending on complex program logic. Unfortunately, Bitcoin – the largest and by far most widely used cryptocurrency – does not offer support for complex smart contracts. Moreover, simple contracts that can be executed on Bitcoin are often cumbersome to design and very costly to execute. In this work we present FastKitten, a practical framework for executing arbitrarily complex smart contracts at low costs over decentralized cryptocurrencies which are designed to only support simple transactions. To this end, FastKitten leverages the power of trusted computing environments (TEEs), in which contracts are run off-chain to enable efficient contract execution at low cost. We formally prove that FastKitten satisfies strong security properties when all but one party are malicious. Finally, we report on a prototype implementation which supports arbitrary contracts through a scripting engine, and evaluate performance through benchmarking a provably fair online poker game. Our implementation illustrates that FastKitten is practical for complex multi-round applications with a very small latency. Combining these features, FastKitten is the *first* truly practical framework for complex smart contract execution over Bitcoin.

1 Introduction

Starting with their invention in 2008, decentralized cryptocurrencies such as Bitcoin [54] currently receive broad attention both from academia and industry. Since the rise of Bitcoin, countless new cryptocurrencies have been launched to address some of the shortcomings of Nakamoto's original proposal. Examples include Zerocash [50] which improves on Bitcoin's limited anonymity, and Ethereum [17] which offers complex smart contract support. Despite these developments, Bitcoin still remains by far the most popular and intensively studied cryptocurrency, with its current market capitalization of $109 billion which accounts for more than 50% of the total cryptocurrency market size [2].

A particular important shortcoming of Bitcoin is its limited support for so-called smart contracts. Smart contracts are (partially) self-enforcing protocols that allow emitting transactions based on complex program logic. Smart contracts enable countless novel applications in, e.g., the financial industry or for the Internet of Things, and are often quoted as a glimpse into our future [9]. The most prominent cryptocurrency that currently allows to run complex smart contracts is Ethereum [17], which has been designed to support Turing complete smart contracts. While Ethereum is continuously gaining popularity, integrating contracts directly into a cryptocurrency has several downsides as frequently mentioned by the advocates of Bitcoin. First, designing large-scale secure distributed systems is highly complex, and increasing complexity even further by adding support for complex smart contracts also increases the potential for introducing bugs. Second, in Ethereum, smart contracts are directly integrated into the consensus mechanics of the cryptocurrency, which requires in particular that all nodes of the decentralized system execute all contracts. This makes execution of contracts very costly and limits the number and complexity of applications that can eventually be run over such a system. Finally, many applications for smart contracts require confidentiality, which is currently not supported by Ethereum.

There has been significant research effort in addressing these challenges individually. Some works aim to extend the functionality of Bitcoin by showing how to build contracts over Bitcoin by using multiparty computation (MPC) [40, 41, 43], others focus on achieving privacy-preserving contracts (e.g., Hawk [38], Ekiden [21]) by combining existing cryptocurrencies with trusted execution environments (TEEs). However, as we elaborate in Section 2, all of these solutions suffer from various deficiencies: they cannot be integrated into existing cryptocurrencies such as Bitcoin, are highly inefficient (e.g., they use heavy cryptographic techniques such as non-interactive zero-knowledge proofs or general MPC), do not support money mechanics, or have significant financial costs due to complex transactions and high collateral (money blocked by the parties in MPC-based solutions).

In this work, we propose FastKitten, a novel system that leverages trusted execution environments (TEEs) utilizing well-established cryptocurrencies, such as Bitcoin, to offer full support for arbitrary complex smart contracts. We emphasize that FastKitten does not only address the challenges discussed above, but is also highly efficient. It can be easily integrated into existing cryptocurrencies and hence is ready to use today. FastKitten achieves these goals by using a TEE to isolate the contract execution inside an enclave, shielding it from potentially malicious users. The main challenges of this solution, such as for instance how to load and validate blockchain data inside the enclave or how to prevent denial of service attacks, are discussed in Section 3.1. Moving the contract execution into the secure enclave guarantees correct and private evaluation of the smart contract even if it is not running on the blockchain and verified by the decentralized network. This approach circumvents the efficiency shortcoming of cryptocurrencies like Ethereum, where contracts have to be executed in parallel by thousands of users. Most related to our work is the recently introduced Ekiden system [21], which uses a TEE to support execution of multiparty computations but does support contracts that handle coins. While Ekiden is efficient for single round contracts, it is not designed for complex reactive multi-round contracts, and their off-chain execution. The latter is one of the main goals of FastKitten.

We summarize our main goals and contributions below.

- **Smart Contracts for Bitcoin:** We support arbitrary multi-round smart contracts executed amongst any finite number of participants, where our system can be run on top of any cryptocurrency with only limited scripting functionality. We emphasize that Bitcoin is only one example over which our system can be deployed today; even cryptocurrencies that are simpler than Bitcoin can be used for FastKitten.
- **Efficient Off-Chain Execution:** Our protocol is designed to keep the vast majority of program execution off-chain in the standard case if all parties follow the protocol. Since our system incentivizes honest behavior for most practical use cases, FastKitten can thus run in real-time at low costs.
- **Formal Security Analysis:** We formally analyze the security of FastKitten in a strong adversarial model. We prove that either the contract is executed correctly, or all honest parties get their money back that they have initially invested into the contract, while a malicious party loses its coins. Additionally, the service provider who runs the TEE is provably guaranteed to

not lose money if he behaves honestly.

- **Implementation and benchmarking:** We provide an in-depth analysis of FastKitten's performance and costs and evaluate our framework implementation with respect to several system parameters by offering benchmarks on real-world use cases. Concretely, we show that online poker can run with an overall match latency of 45ms and costs per player are in order of magnitude of one USD, which demonstrates FastKitten's practicality.

We emphasize that FastKitten requires only a single TEE which can be owned either by one of the participants or by an external service provider which we call the *operator*. In addition, smart contracts running in the FastKitten execution framework support private state and secure inputs, and thus, offer even more powerful contracts than Ethereum. Finally, we stress that FastKitten can support contracts that may span over multiple different cryptocurrencies where each participant may use her favorite currency for the money handled by the contract.

2 Related Work

Support for execution of arbitrary complex smart contracts over decentralized cryptocurrencies was first proposed and implemented by the Ethereum cryptocurrency. As pointed out in Section 1, running smart contracts over decentralized cryptocurrencies results in significant overheads due to the replicated execution of the contract. While there are currently huge research efforts aiming at reducing these overheads (for instance, via second layer solutions such as state channels [52, 26], Arbitrum [36] or Plasma [58], outsourcing of computation [61], or permissioned blockchains [49]), these solutions work only over cryptocurrencies with support complex smart contracts, e.g. over Ethereum. Another line of work, which includes Hawk [39] and the "Ring of Gyges" [35], is addressing the shortcoming that Ethereum smart contracts cannot keep private state. However, also these solutions are based on complex smart contracts and hence cannot be integrated into popular legacy cryptocurrencies such as Bitcoin, which is the main goal of FastKitten.

In this section we will focus on related work, which considers smart contract execution on Bitcoin. We separately discuss multiparty computation based smart contracts and solutions using a TEE. We provide a more detailed discussion on how the above-mentioned Ethereum based solutions compare to FastKitten in Appendix A. Additionally, in Section 8 we discuss some exemplary contract use cases and compare their execution inside FastKitten with the execution over Ethereum.

Approach	**Minimal # TX**	**Collateral**	**Generic Contracts**	**Privacy**
Ethereum contracts	$\mathcal{O}(m)$	$\mathcal{O}(n)$	✓	✗
MPC [42, 43, 41]	$\mathcal{O}(1)$	$\mathcal{O}\left(n^2 m\right)$	✓	✓
[21, 12, 37]	$\mathcal{O}(m)$	no support for money		✓
FastKitten	$\mathcal{O}(1)$	$\mathcal{O}(n)$	✓	✓

Table 1: Selected solutions for contract execution over Bitcoin and their comparison to Ethereum smart contracts. Above, n denotes the number of parties and m is the number of reactive execution rounds.

Multiparty computation for smart contracts An interesting direction to realize complex contracts over Bitcoin is to use so-called multiparty computation with penalties [42, 43, 41]. Similar to FastKitten these works allow secure m-round contract execution but they rely on the claim-or-refund functionality [42]. Such a functionality can be instantiated over Bitcoin and hence these works illustrate feasibility of generic contracts over Bitcoin. Unfortunately, solutions supporting generic contracts require complex (and expensive) Bitcoin transactions and high collateral locked by the parties which makes them impractical for most use-cases. Concretely, in all generic n-party contract solutions we are aware of, each party needs to lock $\mathcal{O}(nm)$ coins, which overall results in $\mathcal{O}(n^2m)$ of locked collateral. In contrast, the total collateral in FastKitten is $\mathcal{O}(n)$, see column "Collateral" in Table 1.

It has been shown that for specific applications, concretely, a multi-party lottery, significant improvements in the required collateral are possible when using MPC-based solutions [51]. This however comes at the cost of an inefficient setup phase, communication complexity of order $\mathcal{O}(2^n)$, and $\mathcal{O}(\log n)$ on-chain transactions for the execution phase. Let us stress that the approach used in [51] cannot be applied to generic contracts.

Overall, while MPC-based contracts are an interesting direction for further research, we emphasize that these systems are currently far from providing a truly practical general-purpose platform for contract execution over Bitcoin—which is the main goal of FastKitten.

TEEs for blockchains There has recently been a large body of work on using TEEs to improve certain applications on blockchains [67, 68, 10, 46, 62]. A prominent example is Teechain [46], which enables off-chain payment channel systems over Bitcoin. However, like most of these prior works, Teechain does not use the TEE for smart contract execution. Some other works, including Hawk [39] and the "Ring of Gyges" [35], propose privacy preserving off-chain contracts execution using TEEs, but do not work over Bitcoin. We refer the reader to Appendix A for more details.

Probably most related to our work are [21, 12, 37], that propose blockchain agnostic systems for private off-chain function execution using TEEs. Despite the conceptual similarities of these works and FastKitten, the goals are orthogonal. The goal of the above mentioned works is to move heavy computation off the chain in order to reduce the cost of executing complex contract functions. However, they do not aim to reduce the communication load on the blockchain. In fact, the communication complexity is often increased compared to the naive on-chain execution of the contracts. In contrast, FastKitten aims to minimize the on-chain communication (especially for multi round applications) and hence can be viewed as a full-fledged blockchain scaling solution.

The works [21, 12, 37] consider clients (contract parties) and computing nodes which have a similar task as FastKitten's TEE operator since they also execute contracts inside a TEE. In contrast to FastKitten, they send the encryption of the resulting contract state to the blockchain after every function call. If a client requests another function call, a selected computing node takes the state from the blockchain, decrypts it inside its enclave and performs the contract execution. This implies that reactive multi-round contracts are very costly even in the standard case when all participating parties are honest (c.f. column "Minimal # TX" in Table 1). All works [21, 12, 37] rely on multiple TEEs to guarantee service availability as long as at least one TEE is controlled by an honest computing node. Even though FastKitten only relies on a single TEE operator, we discuss in Section 9.2 how fault tolerance can be integrated into the system in a straightforward way. A joint goal of all systems is to provide state privacy of the contracts.

Since one of the main goals of FastKitten is to provide a scalability solution of multi-round applications, it incentivices parties to minimize the blockchain interaction as much as possible. Fair distribution of coins is guaranteed through penalizing malicious parties. Since this is not the focus of [21, 12, 37] fair coin distribution is not discussed. For example, Ekiden does not even model or discuss the handling of coins. It is not straightforward to add this feature to their model since the contract state is encrypted and hence the money cannot be unlocked automatically on-chain.

The works [21, 12, 37] are independent projects that have similar goals and approaches. The main difference between Private Data Object [12], compared to Ekiden [21] is the way keys are distributed among TEEs. In [12], the owner of a contract can decide himself which computing nodes get access to the decryption key needed for the contract state decryption. This is in contrast to Ekiden, where all computing nodes have the access rights by default. On the other hand, Ekiden aims to achieve forward secrecy even if a small fraction of TEEs gets corrupted via, e.g., a side-channel attack. Their strategy is to secret-share a long-term secret key between the TEEs and use it to generate a short-term secret key every "epoch". Hence, an attacker learning the short-term key can only decrypt state from the current epoch[1]. The work of [37] provides a more generic construction for using blockchain to achieve statefulness and connectivity of TEEs compared to [21, 12]. In addition it provides a formal model, a rigorously security analysis and discusses multiple applications, like private smart contracts or fairness in multi-party computation.

3 Design

FastKitten allows a set of n users $P_1, \ldots, P_n$ to execute an arbitrary complex smart contract over a decentralized cryptocurrency that only supports very simple scripts. Concretely, FastKitten considers cryptocurrencies that, in addition to supporting simple transactions between users, offer so-called *time-locked transactions*. A transaction is time-locked if it is only processed and integrated into the blockchain after a certain amount of time has passed. Moreover, FastKitten requires that transactions contain space for storing arbitrary raw data. We emphasize that these are very mild requirements on the underlying cryptocurrency that, for instance, are satisfied by the most prominent cryptocurrency Bitcoin.[2] FastKitten leverages these properties together with the power of trusted execution environments to provide an efficient general-purpose smart contract execution platform.

As discussed in the introduction, a contract is a program that handles coins according to some—possibly complex—program logic. In this work, we consider n-party contracts, which are run among a group of parties $P_1, \ldots, P_n$ and have the following structure. During the initialization phase, the contract receives coins from the parties and some initial inputs. Next, it runs for m reactive rounds, where in each round the contract can receive additional inputs from the parties P_i, and produces an output. Finally, after the m-th round is completed the contract pays out the coins to the parties according to its final state and terminates.

A key feature of FastKitten is very low execution cost and high performance compared to contract execution over cryptocurrencies such as Ethereum. This is achieved by not executing contracts *by all parties maintaining the cryptocurrency* but instead running the contract within a single TEE which is operated by a party which we call the *operator* Q. In practice this operator will either be one of the users or a designated service provider, that takes a small fee for his service. In particular this also means that not every participating party that wants to run a contract needs to own a TEE itself, making the system easy accessible for everyone.

In the standard case when all parties are honest, FastKitten runs the entire contract off-chain within the enclave and only needs to touch the blockchain during contract initialization and finalization. More concretely, during initialization, the parties transfer their coins to the enclave by time-locking coins with *deposit transactions*, while at the end of finalization the enclave produces transactions that transfer coins back to the users according to the results of the contract execution. These transactions are called *output transactions* and can be published by the users of the system to receive their coins.

[1] While side-channel attacks are out of scope of this work, note that FastKitten can achieve forward secrecy of states in case of side-channel attacks using the same mechanism as Ekiden.

[2] Bitcoin transactions can store up to 97 KB of data [47]; multiple transactions can be used for bigger payloads.

3.1 Design Challenges of FastKitten

Leveraging TEEs for building a general-purpose contract execution platform requires us to resolve the following main challenges.

Protection against malicious operator. The operator runs the TEE and hence controls its interaction with the environment (e.g., with other parties or the blockchain). Thus, the operator can abort the execution of the TEE, delay and change inputs, or drop any ingoing or outgoing message. To protect honest users from such an operator, the enclave program running inside the TEE must identify such malicious behavior and punish the operator. In particular, we require that even if the TEE execution is aborted, all parties must be able to get their coins refunded eventually. To achieve this, we let the operator create a so-called *penalty transaction*: the penalty transaction time-locks coins of the operator, which in case of misbehavior can be used to refund the users and punish the operator.

Note that designing such a scheme for punishment is highly non-trivial. Consider a situation where party P_i was supposed to send a message x to the contract. From the point of view of the enclave that runs the contract, it is not clear whether the operator was behaving maliciously and did not forward a message to the enclave, or, e.g., party P_i did not send the required message to the operator. To resolve this conflict, we leverage a challenge-response mechanism carried out via the blockchain. We emphasize that this challenge-response mechanism is only required when parties are malicious, and typically will not be executed often due to the high financial costs for an adversary.

Verification of blockchain evidence. To ensure that a malicious operator cannot make up false blockchain evidence, we need to design a secure blockchain validation algorithm which can efficiently be executed inside a TEE. We achieve this by simplifying the verification process typically carried out by full blockchain nodes by using a *checkpoint block* to serve as the initial starting point for verification. This drastically reduces blockchain verification time in comparison to verification starting from the genesis block. To further speed up the transaction verification, we only validate correctness of block headers. Finally, when the TEE needs to verify whether a certain transaction was integrated into a block, we set a minimum number of blocks that must confirm a transaction as part of the security parameter within our protocol. This guarantees that faking a valid-looking chain is computationally infeasible for a malicious operator. Finally, it is computationally infeasible for a malicious operator to load a fake (but valid-looking) chain into the enclave before the penalty transaction is published on the blockchain.

Minimizing blockchain interaction. Since blockchain interactions are expensive, FastKitten only requires interaction with the blockchain in the initialization and finalization phases if all parties follow the protocol. As already discussed above, however, in case of malicious behavior FastKitten may require additional interaction with the blockchain for conflict resolution. This is required to allow the TEE to attribute malicious behavior either to the operator or to some other participant P_i that provides input to the contract. We achieve this through a novel challenge-response protocol, where the TEE will ask the operator to challenge P_i via the blockchain. The operator can then either deliver a proof that he challenged P_i via the blockchain but did not receive a response, in which case P_i will get punished; or the operator receives P_i's input and can continue with the protocol.

Of course, this challenge-response protocol adds to the worst-case execution time of our system, and additionally will result in fees for blockchain interaction. To address the latter, our protocol ensures that both parties involved in the challenge-response mechanism have to split the fees resulting from blockchain interaction equally.[3] This incentivizes honest behavior if parties aim to maximize their

[3] In the cryptocurrency community, this is often referred to as griefing factor 1 : 1, meaning that for every coin spent by the honest users on fees the adversary is required to also spend one coin.

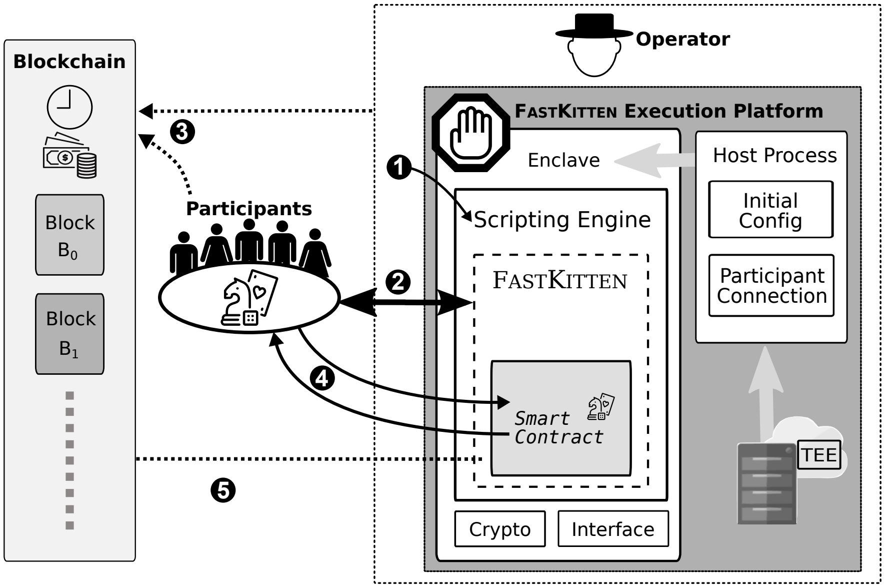


Figure 1: Architecture of the **FastKitten** Smart Contract Execution Platform. Dashed arrows indicate interaction with the blockchain and non-dashed arrows depict communication between parties.

personal profits.

Preventing denial of service attacks. Complex smart contracts may take a very long time to complete, and in the worst case not terminate. Hence, a malicious party may carry out a denial-of-service attack against the contract execution platform, where the platform is asked to execute a contract that never halts. It is well known that determining whether a program terminates is undecidable. Hence, general-purpose contract platforms, such as Ethereum, mitigate this risk by letting users pay via fees for every step of the contract execution. This effectively limits the amount of computation that can be carried out by the contract. Since FastKitten allows multiple parties to provide input to the contract in the same round, it might be impossible to decide which party (parties) caused the denial of service and should pay the fee. To this end, FastKitten protects against such denial-of-service attacks using a time-out mechanism. As all users of the system (including the operator) have to agree on the contract to be executed, we assume that this agreement includes a limit on the maximum amount of execution steps that can be performed inside the enclave per one execution round. See Section 6.5 for more details.

3.2 Architecture and Protocol

To enable secure off-chain contract execution, our architecture builds on existing TEEs, which are widely available through commercial off-the-shelf hardware. In particular, our architecture can be implemented using Intel's Software Guard Extensions (SGX) [48, 31, 4] which is a prominent TEE instantiation built into most recent Intel processors. SGX incorporates a set of new instructions to create, control and communicate with enclaves. While enclaves are part of a legacy host process, SGX enforces strict isolation of computation and memory between enclave and host process on the hardware level. Another prominent instantiation of the TEE concept is ARM TrustZone [6], which provides similar functionality for mobile devices. We note that only the operator Q is required to own TEE-enabled hardware.

As depicted in Figure 1, our FastKitten Execution Facility is run by the operator Q and consists of a host process and an enclave. The untrusted host process takes care of setting up the enclave with an *initial config*, handles the *participant connections*, and blockchain communication over the network. While this means that Q has complete control over these parts, the influence of a malicious operator on a running enclave is limited: he can interrupt enclave execution, but not tamper with it. Further, the enclave will sign and hash all code and data as part of its *attestation* towards parties, so they can verify correctness of the setup before placing deposits. To support arbitrary contract functionality, FastKitten includes a *scripting engine* inside the enclave and several helper libraries, such as the *Crypto library* to generate and verify transactions, and an *Interface library* to pass data between host process and enclave. The individual contracts are loaded into the FastKitten enclave during the initialization of our protocol by the underlying host process and participants can verify that contracts are loaded correctly. Our protocol then proceeds in three phases, which we call *setup phase*, *round computation*, and *finalization phase*. Figure 1 depicts the architecture of the FastKitten framework.

During the setup phase (Steps ❶–❸) the contract is loaded into the enclave. Using the TEE's attestation functionality, all parties $P_1, \ldots, P_n$ can verify that this step was completed correctly. Then the operator and all parties block their coins for the contract execution. If any party aborts in this phase, the money is refunded to all parties that deposited money and the protocol stops. Otherwise, all parties receive a time-locked penalty transaction, needed in case Q aborts the protocol. Afterwards, the round computation phase (Step ❹) starts, in which Q sends the previous round's output to all parties. If a party P_i receives such an output, which is correctly signed by the enclave, it signs and sends the input for the following round to Q. If all parties behave honestly, Q will forward the received round inputs to the enclave, which computes the outputs for the next round. In case that the enclave does not receive an input from party P_i the enclave needs to determine whether P_i failed to send its input or if Q behaved maliciously (e.g., by dropping the message). Therefore, the enclave will punish Q unless it can prove, that it sent the last round output to P_i but did not receive a response. This proof is generated via the blockchain: Q publicly challenges P_i to respond with the input for the next round by posting the output of the previous round to the blockchain. As soon as this challenge transaction is confirmed, P_i needs to respond publicly by spending the coins of the challenge transaction and include its input for the next round. If P_i responds, Q can extract P_i's input and continue with the protocol execution. If P_i did not respond, Q forwards the respective blocks as a transcript to the enclave, to prove that P_i misbehaved.[4] So, while a malicious party (or the operator) can force this on-chain challenge-response procedure without direct punishment, posting these transactions will also act against its own financial interests by extending the time lock of its own coins and leading to transaction fees. Nevertheless, such malicious behavior cannot prevent the fair termination of our protocol.

The last phase of the protocol is the payout phase (Step ❺). In this phase the enclave returns the output transaction generated by the *Crypto library*. This transaction distributes the coins according to the terminated contract. In case of a protocol abort, the coins initially put by the users will be refunded to all honest parties. If any party was caught cheating, this party will not receive back its coins. This means the money will stay in control of the enclave and will never be spent.

4 Adversary Model

The FastKitten protocol is executed n parties $P_1, \ldots, P_n$ and an operator Q (who owns the TEE) with the goal of executing a smart contract C. FastKitten's design depends on a TEE to ensure its confidentiality and integrity. Our design is TEE-agnostic, even if our implementation is based on

[4] Alternatively, we could allow the operator to spend the challenge transaction after a timeout has passed. While this would result in easier verification for the TEE, the operator would need to publish an additional transaction, increasing both fees and the overall time for the challenge-response phase.

Intel SGX. Recent research showed that the security and privacy guarantees of SGX can be affected by memory-corruption vulnerabilities [11], architectural [14] and micro-architectural side-channel attacks [63]. For the operator, we assume that Q has full control over the machine and consequently can execute arbitrary code with supervisor privileges. While memory corruption vulnerabilities can exist in the enclave code, a malicious operator must exploit such vulnerabilities through the standard interface between the host process and the enclave. For the enclave code, we assume a common code-reuse defense such as control-flow integrity (CFI) [3, 16], or fine-grained code randomization [25, 45] to be in place and active. Architectural side-channel attacks, e.g., based on caches, can expose access patterns [14] from SGX enclaves (and therefore our FastKitten prototype). However, this prompted the community to develop a number of software mitigations [60, 29, 20, 13, 59] and new hardware-based solutions [55, 24, 30]. Microarchitectural side-channel attacks like Foreshadow [63] can extract plaintext data and effectively undermine the attestation process FastKitten relies on, leaking secrets and enabling the enclave to run a different application than agreed on by the parties; however, the vulnerability enabling Foreshadow was already patched by Intel [34]. Since existing defenses already target SGX vulnerabilities and since FastKitten's design is TEE agnostic (i.e., it can also be implemented using ARM TrustZone or next-generation TEEs), we consider mitigating side-channel leakage as an orthogonal problem and out of scope for this paper.

For our protocol we consider a *byzantine adversary* [44], which means that corrupted parties can behave arbitrarily. In particular, this includes aborting the execution, dropping messages, and changing their inputs and outputs even if it means that they will lose money. FastKitten is secure even if n parties are corrupt (including the two cases where only the operator is honest, and only one party is honest but the operator is corrupt). We show that no honest party will lose coins, a corrupt party will be penalized and that no adversary can tamper with the result of the contract execution. While we prove security in this very strong adversarial model, we additionally observe that incentive-driven parties (i.e., parties that aim at maximizing their financial profits) will behave honestly, which significantly boosts efficiency of our scheme.

We stress that security of FastKitten relies on the security of the underlying blockchain. We require that the underlying blockchain systems satisfies three security properties: *liveness*, *consistency* and *immutability* [28]. *Liveness* means that valid transactions are guaranteed to be included within the next δ blocks. *Consistency* guarantees that eventually all users have the same view on the current state of the blockchain (i.e., the transactions processed and their order). In addition, blockchains also are *immutable*, which means that once transactions end up in the blockchain they cannot be reverted. Most blockchain based cryptocurrencies guarantee consistency and immutability only after some time has passed, where time is measured by so-called *confirmations*. A block b_i is confirmed k-times if there exists a valid chain extending b_i with k further blocks. Once block b_i has been sufficiently often confirmed, we can assume that the transactions in b_i cannot be reverted and all honest parties agree on an order of the chain $(b_0, b_1, b_2, \ldots, b_i)$. For most practical purposes k can be a small constant, i.e., in Bitcoin it is generally believed that for $k = 6$ a block can be assumed final.[5]

5 The FastKitten Protocol

In this section we give a more detailed description of our protocol, which includes the specification of the protocol run by Q and honest parties $P_1, \ldots, P_n$, all transactions and a description of the

[5] We notice that in blockchain-based cryptocurrencies there is no guaranteed finality, and even for very large values of k blocks can be reverted in principle. We emphasize however that even for small values of k reverting blocks becomes impossible in practice very quickly.

enclave program FastKitten. The interaction between Q, P_i and the blockchain is depicted in Figure 2. We first describe the interactions with the blockchain and TEE.

5.1 Modeling the Blockchain

We will introduce some basic concepts of cryptocurrencies that are relevant for our work before we describe our high-level design. Cryptocurrencies are built using blockchains—a distributed data structure that is maintained by special parties called *miners*. The blockchain is comprised as a chain of blocks $(b_0, b_1, b_2, \ldots)$ that store the transactions of the system. The miners create new blocks by verifying new transactions and comprising them into new blocks that extend the tail of the chain. New blocks are created within some period of time t, where, for instance, in Bitcoin a new valid block is created every 10 minutes on average.

In cryptocurrencies users are identified by addresses, where an address is represented by a public key. To send coins from one address to another, most cryptocurrencies rely on transactions. If a user A with address pk_A wants to send x coins to user B with address pk_B, she creates a transaction tx which states that x coins from address pk_A are transferred to pk_B. Such a transaction tx is represented by the following tuple:

$$\mathsf{tx} := (\mathsf{tx.Input}, \mathsf{tx.Output}, \mathsf{tx.Time}, \mathsf{tx.Data}),$$

where tx.Input refers to a previously unspent transaction, tx.Output denotes the address to which tx.Value are going to be transferred to. Note that a transaction tx is unspent if it is not referred to by any other transaction in its Input field. Further, $\mathsf{tx.Time} \in \mathbb{N}$, which denotes the block counter after which this transaction will be included by miners, i.e., tx can be integrated into blocks $b_i, b_{i+1}, ...$, where $i = \mathsf{tx.Time}$. Finally, $\mathsf{tx.Data} \in \{0,1\}^*$ is a data field that can store arbitrary raw data. Similar to [5], we will often represent transactions by tables as shown exemplary in the table below, where the first row of the table gives the name of the transaction.

Transaction tx	
tx.Input:	Coins from unspent input transaction
tx.Output:	Coins to receiver address
tx.Time:	Some timelock (optional)
tx.Data:	Some data (optional)

Notice that a transaction tx only becomes valid if it is signed with the corresponding secret key of the output address from tx.Input. We emphasize that the properties described above are very mild and are for instance achieved by the most prominent cryptocurrency Bitcoin.

In order to model interaction with the cryptocurrency, we use a simplified blockchain functionality BC, which maintains a continuously growing chain of blocks. Internally it stores a block counter c which starts initially with 0 and is increased on average every t minutes. Every time the counter is increased, a new block will be created and all parties are notified. To address the uncertainty of the block creation duration we give the adversary control over the exact time when the counter is increased but it must not deviate more than $\Delta \in [t-1]$ seconds from t. Whenever any party publishes a valid transaction, it is guaranteed to be included in any of the next δ blocks.

Parties can interact with the blockchain functionality BC using the following commands.

- BC.post(tx): If the transaction tx is valid (i.e., all inputs refer to unspent transactions assigned to creator of tx and the sum of all output coins is not larger than the sum of all input coins) then tx is stored in any of the blocks $\{b_{c+1}, \ldots, b_{c+\delta}\}$.

- $\mathsf{BC.getAll}(i)$: If $i < c$, this function returns the latest block count $c-1$ and a list of blocks that extend b_i: $\mathbf{b} = (b_{i+1}, \ldots, b_c)$
- $\mathsf{BC.getLast}()$: The function $\mathsf{getLast}$ can be called by any party of the protocol and returns the last (finished) block and its counter: (c, b_c).

For every cryptocurrency there must exist a validation algorithm for validating consistency of the blocks and transactions therein, which we model using the function $\mathsf{Extends}$. It takes as input, a chain of blocks $\mathbf{b}$ and a checkpoint block b_{cp} and outputs 1 if $\mathbf{b} = (b_{\mathsf{cp}+1}, \ldots, b_{\mathsf{cp}+i})$ is a valid chain of blocks extending b_{cp} and otherwise it outputs 0. In Section 6 we give more details on the validation algorithm, and how this function is implemented for the Bitcoin system. Recall, that we assume an adversary which cannot compute a chain of blocks of length k by itself (c.f. Section 4). This guarantees that he cannot produce a false chain such that this function outputs 1. To make the position of some transaction tx inside a chain of blocks explicit, we write $\ell := \mathsf{Pos}(\mathbf{b}, \mathsf{tx})$ when the transaction is part of the ℓ-th block of $\mathbf{b}$. If the transaction is in none of the blocks, the function returns ∞. For more details on the transaction and block verification we refer the reader to [54, 28, 7].

5.2 Modeling the TEE

In order to model the functionality of a TEE, we follow the work of Pass et. al. [57]. We explain here only briefly the simplified version of the TEE functionality whose formal definition can be found in [57, Fig. 1]. On initialization, the TEE generates a pair of signing keys (mpk, msk) which we call master public key and master secret key of the TEE. The TEE functionality has two enclave operations: $\mathsf{install}$ and resume. The operation TEE.$\mathsf{install}$ takes as input a program p which is then stored under an enclave identifier eid. The program stored inside an enclave can be executed via the second enclave operation TEE.resume which takes as input an enclave identifier eid, a function f and the function input in. The output of TEE.resume is the output out of the program execution and a quote ϱ over the tuple (eid, p, out). In the protocol description we abstract from the details how the users verify the quote that is generated through the enclave attestation. Since we only consider one instance E of the specific program p, we will simplify the resume command $[out, \varrho] := \text{TEE}.\mathsf{resume}(eid, \mathsf{f}, in)$ and write[6]:

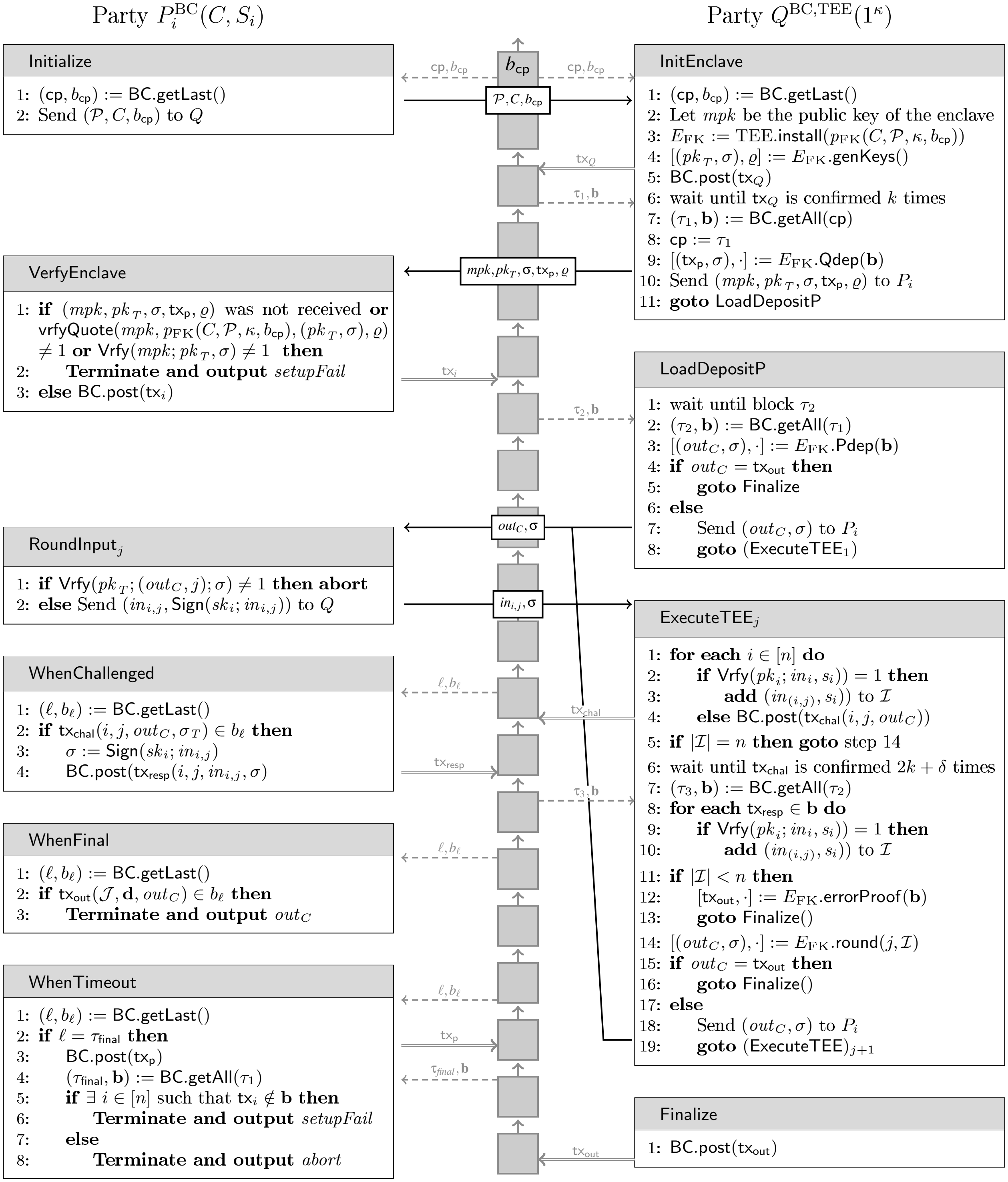
$$[out, \varrho] := E.\mathsf{f}(in)$$

For every attestable TEE there must exist a function $\mathsf{vrfyQuote}(mpk, p, out, \varrho)$ which on input of a correct quote ϱ outputs 1, if and only if out was outputted by an enclave with master public key mpk and which indeed loaded p. Again, we assume that the adversary cannot forge a quote such that the function $\mathsf{vrfyQuote}()$ outputs 1. For more information on how this verification of the attestation is done in practice we refer the reader to [57].

5.3 Detailed Protocol Description

As explained in Section 3, our protocol $\pi_{\text{FastKitten}}$ proceeds in three phases. During the *setup phase* the contract is installed in the enclave, attested, and all parties deposit their coins. Then the *round execution* follows for all m rounds of the interactive contract. When the contract execution aborts or finishes, the protocol enters the *finalize phase*. We now explain all phases and the detailed protocol steps for all involved parties and the operator Q in depth. The detailed interactions as well as the subprocedure of the parties and the operator are displayed in Figure 2, Figure 3 describes the FastKitten enclave program p_{FK}. Overall the protocol requires six different type of transactions.

[6] Since we only need the quote of the first activation of E, we will omit this parameter from there on.


b_{cp}
$\mathsf{cp}, b_{\mathsf{cp}}$
$\mathsf{cp}, b_{\mathsf{cp}}$
$\mathcal{P}, C, b_{\mathsf{cp}}$
tx_Q
$\tau_1, \mathbf{b}$
$mpk, pk_T, \sigma, \mathsf{tx}_{\mathsf{p}}, \varrho$
tx_i
$\tau_2, \mathbf{b}$
out_C, σ
$in_{i,j}, \sigma$
ℓ, b_ℓ
$\mathsf{tx}_{\mathsf{chal}}$
$\mathsf{tx}_{\mathsf{resp}}$
$\tau_3, \mathbf{b}$
ℓ, b_ℓ
ℓ, b_ℓ
tx_{p}
$\tau_{final}, \mathbf{b}$
$\mathsf{tx}_{\mathsf{out}}$

Party $P_i^{\mathrm{BC}}(C, S_i)$

```
Initialize
1: $(\mathsf{cp}, b_{\mathsf{cp}}) := \mathsf{BC.getLast}()$
2: Send $(\mathcal{P}, C, b_{\mathsf{cp}})$ to $Q$
```

```
VerfyEnclave
1: if $(mpk, pk_T, \sigma, \mathsf{tx}_{\mathsf{p}}, \varrho)$ was not received or
   $\mathsf{vrfyQuote}(mpk, p_{\mathrm{FK}}(C, \mathcal{P}, \kappa, b_{\mathsf{cp}}), (pk_T, \sigma), \varrho)$
   $\neq 1$ or $\mathsf{Vrfy}(mpk; pk_T, \sigma) \neq 1$ then
2:     Terminate and output $setupFail$
3: else $\mathsf{BC.post}(\mathsf{tx}_i)$
```

```
RoundInput$_j$
1: if $\mathsf{Vrfy}(pk_T; (out_C, j); \sigma) \neq 1$ then abort
2: else Send $(in_{i,j}, \mathsf{Sign}(sk_i; in_{i,j}))$ to $Q$
```

```
WhenChallenged
1: $(\ell, b_\ell) := \mathsf{BC.getLast}()$
2: if $\mathsf{tx}_{\mathsf{chal}}(i, j, out_C, \sigma_T) \in b_\ell$ then
3:     $\sigma := \mathsf{Sign}(sk_i; in_{i,j})$
4:     $\mathsf{BC.post}(\mathsf{tx}_{\mathsf{resp}}(i, j, in_{i,j}, \sigma)$
```

```
WhenFinal
1: $(\ell, b_\ell) := \mathsf{BC.getLast}()$
2: if $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{d}, out_C) \in b_\ell$ then
3:     Terminate and output $out_C$
```

```
WhenTimeout
1: $(\ell, b_\ell) := \mathsf{BC.getLast}()$
2: if $\ell = \tau_{\mathsf{final}}$ then
3:     $\mathsf{BC.post}(\mathsf{tx}_{\mathsf{p}})$
4:     $(\tau_{\mathsf{final}}, \mathbf{b}) := \mathsf{BC.getAll}(\tau_1)$
5:     if $\exists\, i \in [n]$ such that $\mathsf{tx}_i \notin \mathbf{b}$ then
6:         Terminate and output $setupFail$
7:     else
8:         Terminate and output $abort$
```

Party $Q^{\mathrm{BC,TEE}}(1^\kappa)$

```
InitEnclave
1: $(\mathsf{cp}, b_{\mathsf{cp}}) := \mathsf{BC.getLast}()$
2: Let $mpk$ be the public key of the enclave
3: $E_{\mathrm{FK}} := \mathrm{TEE}.\mathsf{install}(p_{\mathrm{FK}}(C, \mathcal{P}, \kappa, b_{\mathsf{cp}}))$
4: $[(pk_T, \sigma), \varrho] := E_{\mathrm{FK}}.\mathsf{genKeys}()$
5: $\mathsf{BC.post}(\mathsf{tx}_Q)$
6: wait until $\mathsf{tx}_Q$ is confirmed $k$ times
7: $(\tau_1, \mathbf{b}) := \mathsf{BC.getAll}(\mathsf{cp})$
8: $\mathsf{cp} := \tau_1$
9: $[(\mathsf{tx}_{\mathsf{p}}, \sigma), \cdot] := E_{\mathrm{FK}}.\mathsf{Qdep}(\mathbf{b})$
10: Send $(mpk, pk_T, \sigma, \mathsf{tx}_{\mathsf{p}}, \varrho)$ to $P_i$
11: goto LoadDepositP
```

```
LoadDepositP
1: wait until block $\tau_2$
2: $(\tau_2, \mathbf{b}) := \mathsf{BC.getAll}(\tau_1)$
3: $[(out_C, \sigma), \cdot] := E_{\mathrm{FK}}.\mathsf{Pdep}(\mathbf{b})$
4: if $out_C = \mathsf{tx}_{\mathsf{out}}$ then
5:     goto Finalize
6: else
7:     Send $(out_C, \sigma)$ to $P_i$
8:     goto $(\mathsf{ExecuteTEE}_1)$
```

```
ExecuteTEE$_j$
1: for each $i \in [n]$ do
2:     if $\mathsf{Vrfy}(pk_i; in_i, s_i)) = 1$ then
3:         add $(in_{(i,j)}, s_i))$ to $\mathcal{I}$
4:     else $\mathsf{BC.post}(\mathsf{tx}_{\mathsf{chal}}(i, j, out_C))$
5: if $|\mathcal{I}| = n$ then goto step 14
6: wait until $\mathsf{tx}_{\mathsf{chal}}$ is confirmed $2k + \delta$ times
7: $(\tau_3, \mathbf{b}) := \mathsf{BC.getAll}(\tau_2)$
8: for each $\mathsf{tx}_{\mathsf{resp}} \in \mathbf{b}$ do
9:     if $\mathsf{Vrfy}(pk_i; in_i, s_i)) = 1$ then
10:        add $(in_{(i,j)}, s_i))$ to $\mathcal{I}$
11: if $|\mathcal{I}| < n$ then
12:     $[\mathsf{tx}_{\mathsf{out}}, \cdot] := E_{\mathrm{FK}}.\mathsf{errorProof}(\mathbf{b})$
13:     goto Finalize()
14: $[(out_C, \sigma), \cdot] := E_{\mathrm{FK}}.\mathsf{round}(j, \mathcal{I})$
15: if $out_C = \mathsf{tx}_{\mathsf{out}}$ then
16:     goto Finalize()
17: else
18:     Send $(out_C, \sigma)$ to $P_i$
19:     goto $(\mathsf{ExecuteTEE})_{j+1}$
```

```
Finalize
1: $\mathsf{BC.post}(\mathsf{tx}_{\mathsf{out}})$
```

Figure 2: Protocol $\pi_{\text{FastKitten}}$. Direct black arrows indicate communication between the parties and Q, gray dashed arrows indicate reading from the blockchain and gray double arrows posting on the blockchain.

Setup phase. In the setup phase, each party P_i first runs Initialize to generate its key pairs and gets the latest block b_{cp} which serves as a genesis block or checkpoint of the protocol. Then P_i sends the set of parties $\mathcal{P}$, the b_{cp} and the contract C to the operator Q. Upon receiving the initial values from all n parties, Q runs the subprocedure InitEnclave to initialize the trusted execution of the enclave program $p_{\mathrm{FK}}(\mathcal{P}, C, \kappa, b_{\mathsf{cp}})$ where κ is the security parameter of the scheme. This security parameter κ also determines the values for the timeout period t and the confirmation constant k. This ensures that all parties and the TEE agree on these fixed values. Once p_{FK} is installed in the enclave, it generates key pairs for the protocol execution and in particular the blockchain public key pk_T[7]. Now, Q can make its deposit transaction tx_Q which assigns q coins to the enclave public key.

Q's Deposit Transaction $\mathsf{tx_Q}$	
tx.Input:	Some unspent tx from Q
tx.Output:	Assign q coins to TEE

Let block counter τ_1 denote the time when this transaction has been included and confirmed in the blockchain. Q loads all blocks from cp to τ_1 as evidence to the enclave. If this evidence is correct, the execution of p_{FK} function Qdep outputs a penalty transaction $\mathsf{tx_p}$, stating that after timeout τ_{final} (after which the protocol must be terminated) the q coins of Q's deposit transaction tx_Q are payed out to the parties $P_1, \ldots, P_n$.

Penalty Transaction $\mathsf{tx_p}$	
tx.Input:	Q's Deposit Transaction tx_Q
	For all $i \in [n]$:
tx.Output$_i$:	Assign c_i coins to P_i
tx.Time:	Spendable after τ_{final}

Q sends the penalty transaction to all parties $P_1, \ldots, P_n$, who run subprocedure VerfyEnclave. This transaction is used whenever the protocol does not finish before the final timeout τ_{final}, which equals $(3 + 2m) \times (\delta + k)$ blocks after the protocol start (recall, that we use δ to bound the time until some transaction is guaranteed to be included and it will be confirmed after k blocks).[8] Only if participant P_i received this penalty transaction from Q during the setup and verified that the program $p_{\mathrm{FK}}(\mathcal{P}, C, \kappa, b_0)$ is installed in the enclave, it creates and publishes its deposit transaction.

$\mathbf{P_i}$**'s Deposit Transaction** $\mathsf{tx_i}$	
tx.Input:	Some unspent tx from P_i
tx.Output:	Assign c_i coins to TEE

After time $\tau_2 < \tau_1$, Q executes LoadDepositP and again provides the block evidence to the enclave execution of p_{FK}. If all parties published the deposit transactions, the first-round execution starts. Otherwise the enclave proceeds to the finalize phase and outputs a refund transaction $\mathsf{tx_{out}}(T, \vec{c})$ that returns the deposit back to honest users and Q, where $T \subset \mathcal{P}$ is the set of all parties that submitted the deposit transaction until time τ_2. Note, that the internal state of the contract execution is

[7] For simplicity we omit here, that the enclave might need multiple key pairs for signing transactions and messages.

[8] The definition of τ_{final} guarantees that even if the execution is delayed in every round, an honest operator will not be penalized.

maintained by the p_{FK} program inside the enclave. This guarantees that the contract is not executed on outdated state.

Round computation phase. When the protocol arrives to the round computation phase, Q sends the authenticated output of the enclave to every party P_i and requests input for the next round. Each party P_i runs the round algorithm. Internally it verifies whether the input request came from the enclave by verifying the attached signature. Then it generates and signs its round input and sends it to Q. While P_i waits for the next round, Q verifies all received inputs and their signatures in the $\mathsf{ExecuteTEE}$ subprocedure. If all the parties P_i responded with correctly signed round inputs, Q triggers the execution of the contract in the enclave. Let us emphasize that in this simplified description of our protocol we do not focus on the privacy aspect and hence we omit that all round inputs to the contract could be encrypted with the public key of the enclave. In this case the trusted enclave execution needs to decrypt them before it evaluates the contract on them. See Section 9.3 for more details.

Note that the operator Q may be malicious and refrain from requesting a party P_i for the input to a round computation. Instead Q may pretend that it actually did not receive any input from the party P_i. On the other hand, one can imagine a scenario where Q is behaving honestly but the party P_i is dishonest and does not send the correctly signed round input to Q. Note, that the program p_{FK} cannot distinguish between these two cases without additional information. We will next show how an honest Q can generate a proof to attribute the malicious behavior to P_i. First, Q has to publish a challenge transaction $\mathsf{tx}_{\mathsf{chal}}$ which includes the signed output of the previous step. $\mathsf{tx}_{\mathsf{chal}}$ spends a very small amount μ of coins from Q and assign them to party P_i[9].

Challenge Transaction $\mathsf{tx}_{\mathsf{chal}}(i, j, out_C, \sigma_T)$	
tx.Data:	Store i, j, out_C, σ_T
tx.Input:	Some unspent tx from Q
tx.Output:	Assign μ coins to P_i

Once $\mathsf{tx}_{\mathsf{chal}}$ is included in the blockchain, party P_i can read the correct output information from the transaction. The party should respond with $\mathsf{tx}_{\mathsf{resp}}$, which includes its signed round input. $\mathsf{tx}_{\mathsf{resp}}$ spends the $\mathsf{tx}_{\mathsf{chal}}$ and assigns the μ coins back to Q. The action of P_i is depicted via the $\mathsf{WhenChallenged}$ subprocedure.

Response Transaction $\mathsf{tx}_{\mathsf{resp}}(\mathbf{i}, \mathbf{j}, in, \sigma_{\mathbf{i}})$	
tx.Data:	Store i, j, in, σ_i
tx.Input:	Challenge Transaction $\mathsf{tx}_{\mathsf{chal}}(i, j, state)$
tx.Output:	Assign μ coins to Q

If some party does not send the response after it was challenged, Q can prove this misbehavior to the FastKitten program, by providing the blockchain evidence of the challenge-response transcript. If the enclave program identifies a cheating party, it proceeds to the finalize phase. Otherwise, if all the parties' inputs were received with authentication (possibly after challenge-response phase), Q instructs the enclave to execute the contract on the accumulated input.

The result of the contract execution is the output out_C, the updated state $state$, and a coin distribution denoted by $\mathbf{d}$. If $state$ equals $\perp$, the contract execution is finished, and the protocol proceeds

[9] Cryptocurrencies like Bitcoin allow transactions with very small denominations (e.g. fractions of cents).

to the finalize phase. Otherwise, FastKitten internally stores the state and outputs out_C to Q who sends this output to all parties and waits for next round inputs.

Finalize phase. In the finalize phase, the enclave publishes a final output transaction $\mathsf{tx}_{\mathsf{out}}$ which distributes the coins back to all honest parties. It is parameterized by a set of parties to receive coins $\mathcal{J}$, a final coin distribution $\vec{e}$ and a final state out_C. The transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \vec{e}, out_C)$, spends all deposit transactions tx_i for all $i \in \mathcal{J}$ and Q's deposit transaction tx_Q. It includes the out_C in the data field and assigns q coins back to Q and e_i coins to party P_i, for every $i \in \mathcal{J}$. Let us note that $\mathcal{J} = [n]$ implies correct protocol termination. If $\mathcal{J} \neq [n]$, then some party misbehaved and the protocol failed. Either a party did not make a deposit in the setup phase (signaled by $out_C = \mathit{setupFail}$) or some party aborted in the round computation phase (signaled by $out_C = \mathit{abort}$). In both cases all other parties get their initial deposits back. Note, that if a party P_j is caught cheating by the TEE, it will lose its deposit.

Output Transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \vec{e}, out_C)$	
tx.Data:	Store out_C
tx.Input:	Deposit Transactions $\mathsf{tx}_Q, \{\mathsf{tx}_i\}_{i \in \mathcal{J}}$
tx.Output$_1$:	q coins to Q
	For all $i \in \mathcal{J}$:
tx.Output$_{i+1}$:	e_i coins to P_i

Q now has to publish this transaction to get his coins before time τ_{final} and by that also distributes coins and reveals out_C to honest parties. The participants need to constantly monitor the blockchain for transactions which challenge them or indicate final output. When they see a challenge transaction they respond as described above. If they see an output transaction they know the protocol execution ended and output the final contract output according to subroutine WhenFinal.

6 Execution Facility

As shown in Figure 1, we leverage a TEE for smart contract execution. For our prototype, we implemented FastKitten for the Bitcoin blockchain using Intel SGX as a TEE. We chose Python as our scripting engine because it's memory safe, very well known, and widely available. To interact with the Bitcoin blockchain data in the enclave, we implemented our *Crypto library* using the open-source *breadwallet-core* [15], a simplified payment verification (SPV) library for Bitcoin used by the *Breadwallet* mobile wallet app. To abstract from SGX's peculiarities, and thus simplify smart contract development, we use the Graphene Library OS [19] (referred to as "Graphene" in the rest of the paper) as a basis. Graphene enables running arbitrary native Linux binaries in SGX enclaves while providing compatible library interfaces for networking and other OS services. Note that the design of the FastKitten protocol does not require a trusted time source in the TEE.

6.1 The Enclave Program FastKitten

An execution facility in the sense of FastKitten must provide a set of abstract functionalities like key generation, transaction generation, smart contract execution, and error handling, all executed inside the enclave. This set of procedures is described in detail in Figure 3. We implemented each of the procedures using equivalent Python scripts. It is parameterized by the set of parties $\mathcal{P}$, the contract C which internally specifies the expected deposits $\mathbf{c}$, a security parameter κ and a genesis block b_{cp}. This does not need to be the actual genesis block of the underlying blockchain but it can be a later block which is used as a checkpoint. All parties must verify that this block is indeed

The execution of p_{FK} is initialized with the secret key msk, the set of parties (where every $P_i \in \mathcal{P}$ is identified by its key pk_i), a contract C, a security parameter κ (which also defines the waiting period t and confirm period k) and a checkpoint b_{cp}. Internally it stores the state of the contract $state$ and the status flag s initially set to $state = \emptyset$ and $s = \mathsf{genKeys}$.

```
procedure genKeys()
  if $s \neq \mathsf{genKeys}$ then abort
  $(sk_T, pk_T) := \mathsf{Gen}(1^\kappa)$
  $s := \mathsf{Qdep}$
  return $pk_T, \mathsf{Sign}(msk; pk_T)$

procedure Qdep($\mathbf{b}$)
  if $s \neq \mathsf{Qdep}$ or $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ or $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_Q) > |\mathbf{b}| - k$ then abort
  $s := \mathsf{Pdep}$
  $b_{\mathsf{cp}} :=$ last block of $\mathbf{b}$
  return $\mathsf{tx}_{\mathsf{p}}$                                    ▷ Else, output penalty transaction

procedure Pdep($\mathbf{b}$)
  if $s \neq \mathsf{Pdep}$ or $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ then abort
  set $\mathcal{J} := \emptyset$
  for $i \in \mathcal{P}$ do
      $\ell_i := \mathsf{Pos}(\mathbf{b}, \mathsf{tx}_i)$
      if $\ell_i < \delta$ and $\ell_i < |\mathbf{b}| - k$ then add $i$ to $\mathcal{J}$
  if $\mathcal{J} = [n]$ then
      $s := \mathsf{round}_1$
      $b_{\mathsf{cp}} := \mathbf{b}.\mathsf{last}$
      return $\emptyset, \mathsf{Sign}(sk_T; \emptyset, b_{\mathsf{cp}})$
  else
      $s := \mathsf{terminated}$
      return $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, setupFail)$

procedure round($j, (in_1, \sigma_1) \ldots, (in_n, \sigma_n)$)
  if $s \neq \mathsf{round}_j$ or for any $i \in [n] : \mathsf{Vrfy}(pk_i; in_i, s_i) \neq 1$ then abort
  $(out_C, state', \mathbf{d}) := C(state, \vec{in})$
  if $state' \neq \perp$ then
      $s := \mathsf{round}_{j+1}$
      $state := state'$
      return $out_C, \mathsf{Sign}(sk_T; (out_C, j))$
  else
      $s := \mathsf{terminated}$
      return $\mathsf{tx}_{\mathsf{out}}([n], \mathbf{d}, out_C)$

procedure errorProof, ($j, \mathbf{b}$)
  if $s \neq \mathsf{round}_j$ or $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ then abort
  Let $\sigma := \mathsf{Sign}(sk_T; (out_C, j))$
  $\mathcal{J} := [n]$
  for $i \in \mathcal{P}$ do
      if $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_{\mathsf{chal}}(i, j, out_C, \sigma)) < |\mathbf{b}| - \delta - k$ then
          if $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_{\mathsf{resp}}(i, j, in, \sigma) > |\mathbf{b}| - k$ then
              delete $i$ from $\mathcal{J}$
          else if $\mathsf{Vrfy}(pk_i; in, \sigma) \neq 1$ then
              delete $i$ from $\mathcal{J}$
  $s = \mathsf{terminated}$
  if $\mathcal{J} \neq [n]$ then
      return $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, abort)$
```

Figure 3: FastKitten enclave program $p_{\text{FK}}(\mathcal{P}, C, \kappa, b_{\mathsf{cp}})$

a block of the blockchain. The security parameter κ also determines the waiting time k which is needed for the verification of the blocks.

6.2 Blockchain Verification

Blockchain communication is important for the setup and the finalization phase in the protocol. Thanks to the integrity properties of blockchains, a secure connection between the enclave and the blockchain is not needed if verification of received data can be done in the enclave. As it is not practical to download a complete copy of the blockchain to the enclave, we only concentrate on transactions caused by FastKitten protocol invocation. Thus, it is sufficient to verify that these transactions are part of a valid block—without downloading entire blocks, which can be done efficiently using simplified payment verification (SPV). However, SPV libraries can only prove that a transaction *is* part of a block on the blockchain, but they cannot prove that a transaction *is not* part of any block. As required by the challenge-response case, we added an alternative verification mode that fully downloads every block that could potentially contain the transaction and checks whether its present in any of those blocks.

6.3 Participant Communication

To place the deposits and receive them later, as well for sending input, communication between participants (including the Operator Q) is needed in the off-chain phase. We secure this communication using TLS sockets provided by Python. This transparently encrypts participants' communication, and thus ensures input integrity and confidentiality of parties' messages towards the operator.

6.4 Enclave Setup

In the FastKitten prototype, we leverage Intel SGX as a TEE. SGX is a TEE included in recent Intel CPUs which introduces the concept of isolated hardware *enclaves* that can be created and managed using new CPU instructions. SGX enclaves are even shielded from the operating system; only the CPU is trusted. To support smart contract execution in these enclaves we provide a run-time environment based on Graphene, which replaces the Intel SDK in both the enclave and the host process. This allows Graphene to transparently provide services from the untrusted OS (and check the integrity of the results). To protect the enclave application from the host process, a *manifest* has to be provided at enclave initialization. The manifest includes interfaces, services, and respective integrity checksums, e.g., hashes of files the enclave requires. Accesses to these files will be checked against hashes in the manifest to guarantee integrity.

As depicted by Figure 3, the Execution Facility incorporates a set of functionalities. For key derivation ($genKeys$) we leverage the `rdrand` instruction to get high-entropy randomness inside of the enclave. After checking that tx_Q ($Qdep$) is in the blockchain, the derived private key sk_T is used to generate the penalty transaction tx_p using our Crypto library. tx_p is distributed to the other participants over a TLS connection. Other participants can generate their deposit transactions tx_i ($Pdep$) using a regular wallet. This concludes the setup phase, and the smart contract gets executed ($round$).

The Graphene run-time environment enables FastKitten to support arbitrary Linux binaries, thus, can be used to implement smart contracts. However, instead of allowing binaries, we use a scripting engine based on a Python interpreter in our proof-of-concept implementation. First, this makes development easier for contract developers, as they are not always familiar with lower-level programming languages, and second, this makes smart contracts less prone to memory corruption vulnerabilities. Two use cases we implemented are presented and evaluated in Section 8.

6.5 Denial of Service Protection

The protocol as described in Section 5 assumes instantaneous contract execution meaning that the execution of a contract inside a TEE takes no time. For most practical contracts, this simplifying assumption is reasonable since executing a simple contract function inside a TEE is much faster than the network/blockchain delay. However, this is not true when considering arbitrary contracts which might potentially contain endless loops. Moreover, the halting problem states that it is impossible to predict if a certain algorithm will halt within a certain number of steps. A simple protection against endless loops and denial-of-service attacks, is letting the enclave monitor the execution of the smart contract and terminate execution if the number of execution steps exceeds a predefined limit. If the contract execution is aborted due to an execution timeout, the enclave signs an outputs transaction $\mathsf{tx}_{\mathsf{out}}$ which returns deposited coins back to parties and to the operator.

7 Security

In this section we present the underlying security considerations of FastKitten.
Since our protocol is rather complex, we argue about its security in a model that views a protocol as a stand-alone system. In more robust models, such as for example the Global Universal Composability model [18] which considers concurrent protocol executions and protocol composition, statements and proofs are typically convoluted even for simple protocols.
In order to guarantee security for the protocol, we require three security properties: *correctness*, *fairness* and *operator balance security.*
Intuitively, correctness states that in case all parties behave honestly (including the operator), every party $P_i \in \mathcal{P}$ outputs the correct result and earns the amount of coins she is supposed to get according to the correct contract execution. The fairness property guarantees that if at least one party $P_i \in \mathcal{P}$ is honest, then (i) either the protocol correctly completes an execution of the contract or (ii) all honest parties output *setupFail* and stay financially neutral or (iii) all honest parties output *abort*, stay financially neutral, and at least one corrupt party must have been financially punished. Finally, the operator balance security property says that in case the operator behaves honestly, he cannot lose money.

Theorem 1 (Informal statement). *The protocol $\pi_{\text{FastKitten}}$ as defined in Section 5 satisfies correctness, fairness and operator balance security property.*

The most challenging part of the proof is the fairness property. We need to show how honest parties reach consensus on the result of the execution and prove that coins are always distributed between parties according to this result (even if malicious parties collude with the operator). In order to prove the operator balance security, we show that an honest operator has always enough time to publish a valid output transaction which pays him back his deposit, before the time-locked penalty transaction can be posted on the blockchain.
Incentive-driven adversary If we consider only incentive-driven adversaries, then statement (iii) of the fairness property is never true. Hence, if the setup phase completes successfully, then the result of the protocol is a correct contract execution. This follows directly from the fact, that when the protocol aborts the misbehaving parties lose coins. By definition of incentive-driven parties, losing coins is against their interest. This is why the only possible outcome of the protocol is correct execution of the contract. Moreover, when we consider fees for positing transaction on the blockchain, parties are additionally incentivized to prevent the challenge-response transactions. These additional incentives enforce fast and protocol compliant behavior of the parties.

7.1 Architecture Security

The main goal of FastKitten is to enable efficient execution of general multi-round smart contracts. Hence, we analyze the security of FastKitten with regards to its system architecture and implementation. Possible adversaries can be malicious participants, a malicious operator, or a combination of both.

We note that participating clients are only required to send and receive transactions from the blockchain (e.g., to enter an execution) and the ability to exchange protocol messages (e.g., to play rounds). Hence, client implementations can be based on a diverse set of entirely different code bases in practice, possibly using memory-safe languages such as Python, Go, or Rust. Malicious participants are further limited to interacting with other parties and the operator through the exchange of messages as specified within our protocol, and hence, we focus on the TEE-based execution facility in the following.

A malicious operator could deny execution, however, he is incentivized to adhere to the protocol or lose money. Thus, we assume that the goal of a malicious operator is to try and exploit the execution facility at runtime. Since the operator already controls the host process, the main target would be the enclave that executes the contract. Enclaves have a well-defined interface with the rest of the system, and any attack has to be launched using this interface. By providing fake data through this interface, the attacker could try to exploit a memory-corruption vulnerability in the low-level enclave code to launch (a) a code-reuse attack, e.g., by manipulating enclave stack memory, or (b) a data-only attack, e.g., to leak information about the game state or manipulate Bitcoin addresses in contracts. As mentioned in Section 4, for (a) we assume a standard code-reuse defense such as control-flow integrity [3, 66, 69, 53, 16] or fine-grained code randomization [25, 64, 32, 23, 56, 45]. The core functionality of FastKitten additionally tackles both attack vectors by implementing the main enclave code in Python, which provides memory-safety features such as implicit bounds checking. The only parts that are implemented in unsafe languages are the initialization code of Graphene [19] and the Simple Payment Verification (SPV) library [15]. FastKitten actually has no strong dependency on Graphene in principle, it was mainly used to simplify and speed up prototype implementation. Finally, SPV represents a standard library used by most blockchain clients and an adversary that is able to construct a data-only attack against it would be able to exploit any of those clients connected to the Bitcoin network using the same data-only attack.

8 FastKitten Contracts

In this section we take a look at applications and performance through a number of benchmarks.

8.1 Complexity

The FastKitten protocol consists of *setup*, *round computation* and *finalize* phases. During the *setup* phase, each party P_i deposits a constant amount of coins c_i. The operator needs to deposit an amount $\sum_{i\in[n]} c_i$ which equals the sum of all other deposits from $\mathcal{P}$ together. To post the deposit transactions tx_is and tx_Q, a total of $n+1$ transactions is necessary.

During the *round computation phase*, in the optimistic case FastKitten can operate completely off-chain without any blockchain interaction. Any user can force that challenge response transactions are posted to attribute misbehavior of a party, in any given round. If this (pessimistic) case occurs, it can add 2 to another $2n$ transactions. In the worst case, a challenge response transaction pair needs to be posted on the blockchain for every party P_i at every round $j \in [m]$ leading to $\mathcal{O}(nm)$ blockchain interactions. In *finalize* phase, FastKitten requires one additional payout transaction $\mathsf{tx}_{\mathsf{out}}$ to settle money distribution among parties. Scenarios of missing deposit at the *Setup* phase or an abort by a party at the *round computation* phase are dealt with by posting the refund transaction $\mathsf{tx}_{\mathsf{out}}$ and the penalty transaction tx_{p} respectively.

Setup time In the optimistic case (which we have shown is the standard case when considering incentive-driven parties) the overall execution of the protocol only requires $n+2$ transactions on the blockchain. This also indicates at what speed the protocol can be executed in this case. If all parties agree, the setup phase can be finished in 2 blockchain rounds and from that point on the protocol can be played off-chain. In the next subsection we give some indication how fast this second part can be achieved. Running the protocol as fast as possible is in the interest of every party since it shortens the locking time of the deposits.

8.2 Performance Evaluation

We performed a number of performance measurements to demonstrate the practicality of FastKitten using our lab setup, which consists of three machines: First, an SGX-enabled machine running Ubuntu 16.04.5 LTS with an Intel i7-7700 CPU clocked at 3.60GHz and 8GB RAM, where we installed FastKitten's contract execution facility to play the role of the operator's server. Second, a machine running Ubuntu 14.04.4 LTS on an Intel i7-6700 CPU clocked at 3.40GHz with 32GB RAM, which provides unmodified blockchain nodes in a local test network using Bitcoin Core version 0.16.1. Third, a laptop machine with macOS 10.13.6 on with Intel i7-4850HQ CPU clocked at 2.30GHz and 16GB of RAM, which takes the role of the participants in the protocol. All three machines are connected through a Gigabit Ethernet LAN. For tests involving the real Bitcoin network the individual machines are connected through the Internet using our Internet connection.

Block validation In our experiments, the enclave takes approximately 5 s to validate one block from the Bitcoin main network, thus proving that it is capable of validating real blocks in real time.

Enclave Startup The time to setup an enclave until it is ready is 2 s, proving that instantiating enclaves on the fly is feasible.

End-to-end Time Assuming all parties are incentive-driven and, thus, comply with the protocol, the total time required by FastKitten is the time of 2 blockchain interactions (see Section 8.1), plus the computation time (a few milliseconds in our use cases), plus the time required by the parties to choose the next inputs.

8.3 Applications

FastKitten allows to run complex smart contracts on top of cryptocurrencies that would not natively support such contracts, like Bitcoin. But in contrast to Turing-complete contract execution platforms like Ethereum, a secure off-chain execution such as FastKitten puts some restrictions on the contracts it can run:

- The number of parties interacting with the contract must be known at the start of the protocol.
- It must be possible to estimate an upper bound on the number of rounds and the maximum run time of any round.

All of these restrictions make FastKitten contracts different from smart contracts running on Ethereum itself. The restrictions above come from the fact that the contract can be completely (and repeatedly) executed without blockchain interactions. Other off-chain solutions (like state channels [52, 26, 22]) come with similar caveats. By allowing additional blockchain interaction we could get around those restrictions but we would lose efficiency in the optimistic case (which is also similar to state channel constructions).

FastKitten has important features which are supported by neither Bitcoin nor Ethereum — FastKitten allows private inputs and batched execution of user inputs. Overall, this leads to cheaper, faster and private contract execution than what is possible with on-chain contracts in

Transaction	Size (Bytes)	Fees (BTC)	Fees (USD)
Deposit (typical)	250	0.000007-0.000073	0.05-0.46
Penalty (tx_{p})	504	0.000015-0.000148	0.09-0.93
Challenge ($\mathsf{tx}_{\mathsf{chal}}$)	293	0.000009-0.000086	0.05-0.54
Response ($\mathsf{tx}_{\mathsf{resp}}$)	266	0.000008-0.000078	0.05-0.49
Output ($\mathsf{tx}_{\mathsf{out}}$)	1986	0.000058-0.000582	0.36-3.65

Table 2: Estimated fees for a typical deposit transaction and the FastKitten transactions, using data from CoinMarketCap [2] and BlockCypher [1] retrieved on Nov. 14, 2018.

Ethereum. Below, we highlight these efficiency gains by presenting four concrete use-cases in which FastKitten outperforms contracts run over Ethereum or in Ethereum state channels.

Lottery A lottery contract takes coins from every involved party as input, and randomly selects one winner, who gets all the coins. The key challenge for such a contract is to fairly generate randomness to select the winner. In Ethereum or Bitcoin the randomness is computed from user inputs through an expensive commit-reveal scheme [51]. In FastKitten, all parties can immediately send their random inputs to the enclave which will securely determine a winner. Hence, we reduce the round complexity from $\mathcal{O}(\log n)$ [51] to $\mathcal{O}(1)$.

Auctions Another interesting use-case for smart contracts are auctions, where parties place bids on how much they are willing to pay and the contract determines the final price. In a straightforward auction, the bids can be public, but more fair versions, like second bid auctions, require the users not to learn the other bids before they place their own. The privacy features of FastKitten can be used to reduce the round complexity for such auctions which would otherwise require complex cryptographic protocols [27].

Rock-paper-scissors We implemented the popular two-party game rock-paper-scissors to show the feasibility of FastKitten contracts. Again, the privacy features allow one match to be executed in a single round, which would have required at least 3 rounds in Ethereum. The pure execution time in the optimistic case, excluding delays due to human reaction times, is 12ms for one round (averaged over 100 matches). This demonstrates that off-chain protocols, like FastKitten, are highly efficient when the same set of parties wants to run complex contracts (like multiple matches of a game).

Poker We also implemented a Texas Hold'em Poker game, to prove that multi-party contracts which inherently require multiple rounds can also be efficiently executed in FastKitten. In our implementation, each player starts with an equal chip stack and participates in an initial betting round and in additional rounds after the flop, river, and turn have been dealt by the enclave. If more than two players remain in the game after the final bets, the enclave reveals the winner and distributes the chips in the current pot to the winner. The game continues until only one player remains. We measured 50 matches between 10 players resulting in an average time of 45ms per match (multiple betting rounds are included in each match). The run time was measured starting from the moment all deposits are committed to the blockchain (details on the exact measurements and analysis can be found in the Appendix D).

Real-world Fees We generated examples of the transaction types used in our protocol for a 10-player poker match, which are shown in Appendix E. In Table 2 we estimate the fees required to commit to the blockchain our transactions, in addition to a typical deposit transaction. Assuming all parties comply with the protocol, each party (including Q) must pay between 0.05 USD and 0.46 USD for the deposit. Additionally, the output transaction $\mathsf{tx}_{\mathsf{out}}$ requires between 0.36 USD and 3.65 USD in fees.

Other Well-known Contracts Certain well-known contracts like ERC20 token and CryptoKitties inherently need to be publicly available on the blockchain, since they are accessed frequently by participants which are not previously known. In contrast, contracts resembling our examples above, which rely on private data and where a fixed set of participants sends a large number of transactions, are highly efficient when moved off-chain using a system like FastKitten. The nature of off-chain solutions like FastKitten or state channels requires advance knowledge of the participants. Open contracts like ERC20 and CryptoKitties that require continuous synchronization with the blockchain and are meant to be publicly accessible would eliminate the advantages of off-chain solutions.

9 Discussion and Extensions

In order to explain and analyze the FastKitten protocol, we presented a simplified protocol version which only includes the building blocks required to guarantee security. Depending on the use case one might be interested in further properties. Possible extensions discussed in this section include the option to pay the operator for his service, protect the operator against TEE faults, hide the contract output from through a layer of output encryption and allow cross-currency smart contracts. In the following, we explain how to achieve these features and at what cost they can be added to the simplified protocol.

9.1 Fees for the Operator

The owner of the TEE provides a service to the users who want to run a smart contract and, naturally, he wants to be paid for it. In addition to the costs of buying, maintaining and running the trusted hardware, he also needs to block the security deposit q for the duration of the protocol. While the security of FastKitten ensures that he will never lose this money, he still cannot use it for other purposes. The goal of the operator-fees is to make both investments attractive for Q.

We assume that the operator will be paid ξ coins for each protocol round for each party. Since the maximum number of rounds m is fixed at the protocol start, Q will receive $\xi \times n \times m$ coins if the protocol succeeds (even if the contract terminated in less than m rounds). If the operator proves to the TEE in round x that another party did not respond to the round challenge, he will only receive a fee for the passed x number of rounds (namely $\xi \times x \times n$). This pay-per-round model ensures that the operator does not have any incentive to end the protocol too early. If the protocol setup does not succeed or the operator cheats, he will not receive any coins.

The extended protocol with operator fees requires each party to lock $c_i + m \times \xi$ coins and the operator needs to level this investment with $qc_i + m \times \xi$ coins.

9.2 Fault Tolerance

In order to ensure that the execution of the smart contract can proceed even in the presence of software or hardware faults, the enclave can save a snapshot of the current state in an encrypted format, e.g., after every round of inputs. This encrypted state would be sent to the operator and stored on redundant storage. If the enclave fails, the operator can instantiate a new enclave which will restart the computation starting from the encrypted snapshot. If the TEE uses SGX, snapshots would leverage SGX's sealing functionality [33] to protect the data from the operator while making it available to future enclave instances.

9.3 Privacy

As mentioned in the introduction, traditional smart contracts cannot preserve privacy of user inputs and thus always leak internal data to the public. In contrast to common smart contract technologies, the FastKitten protocol supports privacy preserving smart contracts as proposed in Hawk [39]. This requires *private contract state* to hide the internal execution of the contract and *input privacy*, which means that no party (including the operator) sees any other parties' round input before sending its own.

It is straightforward to see that FastKitten has a secret state, since it is stored and maintained inside the enclave. Input privacy can easily be achieved by encrypting all inputs with the public key of the enclave. This guarantees that only the FastKitten execution facility and the party itself knows the inputs. If required, FastKitten could also be extended to support privacy of outputs from the contract to the parties, by letting the enclave encrypt the individual outputs with the parties' public keys. But this additional layer should only be used when the contract requires it, since in the worst case this increases the output complexity of the challenge and output transaction.

9.4 Multi-currency Contracts

FastKitten requires from the underlying blockchain technology that transactions can contain additional data and can be timelocked. Any blockchain like Bitcoin, Ethereum, Lightcoin and many others which allow these transaction types can be used for the FastKitten protocol. With some minor modifications FastKitten can even support contracts which can be funded via multiple different currencies. This allows parties that own coins in different currencies to still execute a contract (play a game) together. The main modification to the FastKitten protocol is that the operator and the enclave need to simultaneously handle multiple blockchains in parallel. In particular, for each of the considered currencies, Q needs to deposit the sum of all coins that were deposited by parties in that currency. This is in order to guarantee that if the operator cheats, players get back their invested coins in the correct currency. In addition, the operator is obliged to challenge each party via its blockchain. If the execution completes (or the operator proves to the enclave that one of the players cheated), the enclave signs one output transaction for each of the currencies. While this extension adds complexity to the enclave program and leads to more transactions and thus transaction-fees, the overall deposit amount stays identical to the single blockchain use case.[10] A complete design and proof of correctness of a cross-ledger FastKitten is left to future work.

10 Conclusion

In this paper we have shown that efficient smart contracts are possible using only standard transactions by combining blockchain technology with trusted hardware. We present FastKitten, our Bitcoin-based smart contract execution framework that can be executed off-chain. Since FastKitten is the first work that supports efficient multi-round contracts handling coins, for the first time, this enables real-time application scenarios, like interactive online gaming, with millisecond round latencies between participants. We formally prove and thoroughly analyze the security of our general framework, also extensively evaluating its performance in a number of use cases and benchmarks.

Additionally, we discuss multiple extensions to our protocol, such as adding output privacy or operator fees, which enrich the set of features provided by our system.

[10] This solution assumes that any party can receive coins in any of the considered currencies.

Acknowledgments

We are grateful to our anonymous reviewers and our shepherd Mihai Christodorescu for their constructive feedback.
This work has been supported by the German Research Foundation (DFG) as part of projects HWSec, P3 and S7 within the CRC 1119 CROSSING and the Emmy Noether Program FA 1320/1-1, by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by BMBF within the iBlockchain project, by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] BlockCypher, Nov 2018. `https://live.blockcypher.com/btc/`.

[2] CoinMarketCap, Nov 14 2018. `https://coinmarketcap.com`.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.

[4] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[5] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[6] ARM Limited. Security technology: building a secure system using TrustZone technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2008.

[7] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO*, 2017.

[8] J. Barbie. Why smart contracts are not feasible on plasma, Jul 2018. `https://ethresear.ch/t/why-smart-contracts-are-not-feasible-on-plasma/2598`.

[9] G. Belisle. A glimpse into the future of blockchain, 2018. Available at `https://the-blockchain-journal.com/2018/03/29/a-glimpse-into-the-future-of-blockchain/`.

[10] I. Bentov, Y. Ji, F. Zhang, Y. Li, X. Zhao, L. Breidenbach, P. Daian, and A. Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. *IACR Cryptology ePrint Archive*, 2017.

[11] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel sgx. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 2018.

[12] M. Bowman, A. Miele, M. Steiner, and B. Vavala. Private data objects: an overview. *arXiv preprint arXiv:1807.05686*, 2018.

[13] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A. Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017.

[14] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies*, 2017.

[15] Breadwallet. Breadwallet-core - spv bitcoin c library, 2018.

[16] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *CoRR*, 2016.

[17] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[18] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, 2007.

[19] C. che Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference*, 2017.

[20] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *ACM Symposium on Information, Computer and Communications Security*, 2017.

[21] 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 contract execution. *arXiv preprint arXiv:1804.05141*, 2018.

[22] J. Coleman, L. Horne, and L. Xuanji. Counterfactual: Generalized state channels, Jun 2018. https://l4.ventures/papers/statechannels.pdf.

[23] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies*, 2016.

[24] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, 2016.

[25] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2013.

[26] S. Dziembowski, S. Faust, and K. Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018.

[27] H. Galal and A. Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *International Conference on Financial Cryptography and Data Security, Trusted Smart Contracts Workshop. Springer*, 2018.

[28] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *CRYPTO*. Springer, 2017.

[29] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium*, 2017.

[30] M. Hachman. Intel's plan to fix meltdown in silicon raises more questions than answers. https://www.pcworld.com/article/3251171/components-processors/intels-plan-to-fix-meltdown-in-silicon-raises-more-questions-than-answers.html, 2018.

[31] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[32] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: trans-parent code randomization for just-in-time compilers. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2013.

[33] Intel. Intel Software Guard Extensions developer guide, 2016. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf.

[34] Intel. Resources and Response to Side Channel L1 Terminal Fault. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html, 2018.

[35] A. Juels, A. E. Kosba, and E. Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016.

[36] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *USENIX Security Symposium*, 2018.

[37] G. Kaptchuk, I. Miers, and M. a. Green. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *26th Annual Network and Distributed System Security Symposium*, NDSS, 2019.

[38] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.

[39] A. E. 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*, 2016.

[40] R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[41] R. Kumaresan and I. Bentov. Amortizing secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[42] R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[43] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan. Improvements to secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[44] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1982.

[45] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[46] J. Lind, O. Naor, I. Eyal, F. Kelbert, P. R. Pietzuch, and E. G. Sirer. Teechain: Reducing storage costs on the blockchain with offline payment channels. In *11th ACM International Systems and Storage Conference*, 2018.

[47] R. Matzutt, J. Hiller, M. Henze, J. H. Ziegeldorf, D. Müllmann, O. Hohlfeld, and K. Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. In *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC). Springer*, 2018.

[48] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[49] Microsoft. The coco framework, 2018. GIT repository available at `https://github.com/Azure/coco-framework`.

[50] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.

[51] A. Miller and I. Bentov. Zero-collateral lotteries in bitcoin and ethereum. In *Security and Privacy Workshops (EuroS&PW), 2017 IEEE European Symposium on*. IEEE, 2017.

[52] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. Sprites: Payment channels that go faster than lightning. *CoRR*, abs/1702.05812, 2017.

[53] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, 2015.

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

[55] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security symposium*, USENIX Sec, 2013.

[56] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[57] R. Pass, E. Shi, and F. Tramèr. Formal abstractions for attested execution secure processors. *IACR Cryptology ePrint Archive*, 2016.

[58] J. Poon and V. Buterin. Plasma: Scalable autonomous smart contracts, Aug 2017. Plasma, `https://plasma.io/plasma.pdf/`.

[59] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Annual Network and Distributed System Security Symposium*, 2017.

[60] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Annual Network and Distributed System Security Symposium*, 2017.

[61] J. Teutsch and C. Reitwießner. A scalable verification solution for blockchains, Nov 2017. `https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf`.

[62] F. Tramèr, F. Zhang, H. Lin, J. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P*, 2017.

[63] J. Van Bulck, F. Piessens, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[64] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2012.

[65] Worldveil. Deuces-a pure python poker hand evaluation library, 2016.

[66] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[67] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016.

[68] F. Zhang, P. Daian, I. Bentov, and A. Juels. Paralysis proofs: Safe access-structure updates for cryptocurrencies and more. *IACR Cryptology ePrint Archive*, 2018.

[69] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.

A Further Related Work

There is a large body of work trying to improve the scalability of blockchains by moving a major part of smart contract executions off the blockchain (for example, via second layer solutions [52, 58, 26, 36] or outsourcing of computation [61]). As discussed in the main body of this paper, all of these solutions run on top of blockchains with sufficiently complex scripting language, e.g., on Ethereum. However, they cannot be integrated into popular legacy cryptocurrencies such as Bitcoin, which is their main difference compared to our work. Recall that one of the main goals of FastKitten is make minimal assumption on the underlying blockchain technology and in particular, to run over the Bitcoin blockchain.

Another motivation for off-chain contract execution might be the goal of protecting privacy. Hawk [39] and the "Ring of Gyges" [35] are examples of works that do keep the state, all inputs and outputs private. It is also true for the scaling solutions mentioned above; These techniques work only over cryptocurrencies with support for complex smart contracts, e.g. over Ethereum.

Below we discuss the differences between these solutions and FastKitten when run on top of Ethereum.

A.1 Second-layer Scaling Solutions

State Channels State channels [52, 26, 22] are a prominent second layer scaling solution. They allow a set of parties to execute complex smart contracts off-chain. As long as all parties are honest and agree on the state transitions, the blockchain is contacted only during the channel creation, when parties lock funds in the channel, and during channel closure, when the locked funds are distributed back to the parties according to the result of contract execution. However, once parties run into disagreement off-chain, they have to resolve their dispute on-chain and perform the state transition via the blockchain.

While in the optimistic case when all parties are honest, state channels are very efficient, a potentially heavy computation might need to be done on-chain in case of disagreement. This is in contrast to the FastKitten protocol which does not require any computation to be performed on the blockchain even in case of disputes.

Plasma Another promising second-layer scaling solution is Plasma, first introduced by Poon and Buterin [58]. The main idea of Plasma is to build new chains (Plasma chains) on top of the Ethereum blockchain. Each Plasma chain has its own operator that is responsible for validating transactions and regularly posting a short commitment about the current state of the Plasma chain to a smart contract on the Ethereum blockchain. The regular commitments guarantee to the participants of the Plasma chain that in case the operator cheats, his misbehavior can be proven to the Ethereum smart contract and parties can exit the Plasma chain with all their funds.

While the original goal of Plasma [58] was to support arbitrary complex smart contracts, to the best of our knowledge, there is no concrete protocol that would achieve this goal (the existing Plasma designs support only payment transactions). Moreover, the plasma research community currently conjectures that Plasma with general smart contracts might be impossible to construct [8].

A.2 Incentive-driven Verification

Arbitrum The disadvantage of state channels, i.e., the potentially heavy on-chain execution in case of dispute, is being addressed by the work Arbitrum [36]. Every smart contract, which Arbitrum models as a virtual machine (VM), to be executed off-chain has a set of "manager" parties responsible for correct VM execution. As long as managers reach consensus on the VM state transitions, execution progresses off-chain similarly as in state channels. In case of dispute, managers do not perform the VM state transition on-chain as in state channel. Instead, one manager can propose the next VM state which other managers can challenge. If the newly posted state is challenged, the proposer and the challenger run an interactive protocol via the blockchain, so-called "bisection" protocol, in which one disputable computation step is eventually identified and whose correct execution is verified on-chain. Hence, instead of executing the entire state transition on-chain (which might potentially require a lot of time/space), only one computation step of the state transition has to be performed on-chain in addition to the bisection protocol (which might require $\mathcal{O}(\log(s))$ blockchain transactions, where s is the number of computations steps in the state transition). The Arbitrum protocol works under the assumption that at least one manager of the VM is honest and challenges false states if they are posted by other managers. Since the blockchain interaction during the bisection protocol is rather expensive, Arbitrum uses monetary incentives to motivate managers to behave honestly and follow the protocol.

TrueBit Another solution that supports off-chain execution of smart contracts using incentive verification is TrueBit [61]. For each off-chain execution, the TrueBit system selects (using a lottery) one party, called the "Solver", that is responsible for performing the state transition and inform all other parties about the new contract state. The TrueBit system incentives parties to become so called "verifiers" and check the correctness of the computation performed by the Solver. In case they detect misbehavior, they are supposed to challenge the Solver on the blockchain and run the "verification game" which works similarly as the "bisection protocol" of Arbitrum. Similar to Arbitrum, TrueBit relies on the assumption that there is at least one honest verifier which correctly performs all the validations and challenges malicious Solvers. In contrast to Arbitrum, all inputs and the contract state are inherently public even in the optimistic case when everyone is honest.

Apart from the different trust model and lower requirement on the underlying blockchain technology, FastKitten differs from Arbitrum and TrueBit by providing stronger privacy guarantees, meaning that in both the optimistic and the pessimistic case, inputs of honest parties as well as the state of the smart contract remains private.

A.3 TEEs for privacy

None of the solutions discussed above achieves privacy preserving off-chain contract execution. This is tackled by the work Hawk [39] which keeps the state, all inputs and all outputs private. Hawk contracts [38] achieve these properties using Ethereum smart contracts that judge computations done by a third party (a manager), who executes the contract on private inputs and is trusted not to reveal any secrets. First all parties submit their encrypted inputs to the contract, then the manager computes the result and proves its correctness with a zero knowledge proof. If the proof is correct, the contract pays out money accordingly. While the authors of Hawk discuss the possibility to use SGX for instantiating the manager and reducing the trust assumptions in this party, it still leverages the blockchain for every user input, and it only supports single round protocols which is their main difference to FastKitten. A possible extension to multi-round protocols would be difficult to achieve without letting the smart contract verify the correctness of every round individually, and thus create a large blockchain communication overhead.

B Definitions and notation

We denote the set of natural numbers $1, \dots, m$ and $[m]$ and the n-ary Cartesian power of a set $\mathcal{S}$ by $\mathcal{S}^n := \mathcal{S} \times \mathcal{S} \times \cdots \times \mathcal{S}$. We refer to the i-th coordinate of the vector $\mathbf{s} \in \mathcal{S}^n$ as $\mathbf{s}[i]$. An n-by-m matrix of elements from $\mathcal{S}$ is denoted by $\mathbf{S} \in \mathcal{S}^{n \times m}$. The element in the i-th row and j-column of $\mathbf{S}$ is $\mathbf{S}[i][j]$, the i-th row of $\mathbf{S}$ is denoted by $\mathbf{S}[i][\cdot]$ and the j-th column of $\mathbf{S}$ is denoted by $\mathbf{S}[\cdot][j]$.

Modeling coins and contracts Since our framework is not restricted to one specific blockchain, we define a *coin domain* $\mathcal{D}_{coin}$ as a subset of non-negative rational numbers. The concrete definition of the set $\mathcal{D}_{coin}$ depends on the considered blockchain.[11]

In this work we model an n-party contract C as a polynomial time Turing machine. Every contract has to define a vector $\mathbf{c} \in \mathcal{D}_{dep}$ that corresponds to the deposits parties are supposed to make. It must hold that $\mathcal{D}_{dep} = (\mathcal{D}_{coin} \setminus \{0\})^n$ which enforces that every party has to make a deposit. We write $C_\mathbf{c}$ when a reference to the deposit vector is needed. The contract $C_\mathbf{c}$ takes as input a value $\mathit{state} \in \mathcal{D}_{state} \cup \{\emptyset\}$ and a vector of values $\mathbf{in} \in \mathcal{D}_{in}^n$, and returns an output value $\mathit{out} \in \mathcal{D}_{out}$, a new state $\mathit{state} \in \mathcal{D}_{state} \cup \{\bot\}$ and a coin distribution $\mathbf{d} \in \mathcal{D}_{coin}^n$. The initial state of every contract is $\mathit{state} = \emptyset$ and $\mathit{state} = \bot$ signals the contract's termination. In case $\mathit{state} = \bot$, the vector $\mathbf{d}$ defines the final payout to each party of the contract. It must hold that $\sum_{i \in [n]} \mathbf{d}[i] \leq \sum_{i \in [n]} \mathbf{c}[i]$. This restriction guarantees that parties cannot create money by executing a contract.

The input domain $\mathcal{D}_{in}$, the state domain $\mathcal{D}_{state}$ and the output domain $\mathcal{D}_{out}$ are application specific and defined by the contract. For example, in case C is the "Rock-paper-scissor" game, then $\mathcal{D}_{in}$ could be {rock, paper, scissor} and the $\mathcal{D}_{out}$ could be {winA, winB, same}, where the values "winA", "winB" would define the winner and "same" signals that none of the players won.

We model the contract C run among n parties for m rounds. Formally, a n-party contract C is called *m-round bounded* (or shortly an (n, m)-contract) if for any $\mathbf{I} \in \mathcal{D}_{in}^{n \times m}$ there exists $m' \in [m]$ such that $\mathit{state}_{m'} = \bot$, where $(\mathit{state}_j, \mathit{out}, \mathbf{d}) := C_\mathbf{c}(\mathit{state}_{j-1}, \mathbf{I}[\cdot][j])$ for $j \geq 1$ and $\mathit{state}_0 := \emptyset$.

In order to formally state our security properties, we need to define what we mean by *correct evaluation of a contract*. To this end, we define the algorithm Eval in Fig. 4 which takes as input an (n, m)-contract C and a matrix of inputs $\mathbf{I} \in \mathcal{D}_{in}^{n \times m}$. The output of the algorithm is the tuple $(\mathit{out}, \mathbf{d})$, where out is the output and $\mathbf{d}$ is the coin distribution after the contract's termination.

```
$j := 1$, $\mathit{state} := \emptyset$
while $\mathit{state} \neq \bot$ do
    $(\mathit{state}, \mathit{out}, \mathbf{d}) := C(\mathit{state}, \mathbf{I}[\cdot][j])$
    $j := j + 1$
Output $(\mathit{out}, \mathbf{d})$.
```

Figure 4: Correct contract evaluation of an (n, m)-contract C on inputs $\mathbf{I} \in \mathcal{D}_{in}^{n \times m}$.

Protocol execution We consider an $n + 1$ party protocol π that runs between parties from the set $\mathcal{P}^+ = \{P_1, \dots, P_n, P_{n+1}\}$. Sometimes it is convenient to refer to the set of parties without the operator P_{n+1}.[12] To this end we define $\mathcal{P} := \mathcal{P}^+ \setminus \{P_{n+1}\}$. We will slightly abuse the notation

[11] For Bitcoin, for instance, we have $\mathcal{D}_{coin} := \{c \in \mathbb{Q} \mid c \cdot 10^8 \in \mathbb{N} \cup \{0\}\}$ which corresponds to the fact that the smallest Bitcoin unit, called Satoshi, is equal to 10^{-8} BTC.

[12] For simplicity, we denote the operator Q as P_{n+1}

and write $i \in \mathcal{P}^+$ instead of $P_i \in \mathcal{P}^+$. We assume that all parties are connected to the operator by authenticated channels. A protocol is executed in presence of an adversary $\mathcal{A}$ who can *corrupt* parties from $\mathcal{P}^+$. By saying "$\mathcal{A}$ corrupts the party P_i" we mean that the adversary takes complete control over the party P_i (i.e. the adversary learns the inputs of the party P_i, party's internal state and has the power to decide about messages P_i sends and the output P_i returns).
The input of the protocol execution is an (n, m)-contract $C_{\mathbf{c}}$, a matrix of inputs $\mathbf{I} \in \mathcal{D}_{in}^{n \times m}$, where the value $\mathbf{I}[i][j]$ defines the input of party P_i for round j, and a vector of account balances $\mathbf{acc}^{\mathsf{old}} \in \mathcal{D}_{acc}(\mathbf{c}) \subseteq \mathcal{D}_{coin}^{n+1}$. The account domain $\mathcal{D}_{acc}(\mathbf{c})$ is defined such that $\forall i \in \mathcal{P}$: $\mathbf{acc}^{\mathsf{old}}[i] \geq \mathbf{c}[i]$ and $\mathbf{acc}^{\mathsf{old}}[i+1] \geq \sum_{i \in \mathcal{P}} \mathbf{c}[i]$. This restriction guarantees that every party has enough coins for the initial deposit. The output of the execution of the protocol π in presence of an adversary $\mathcal{A}$ is defined as

$$(\mathbf{out}, \mathbf{acc}^{\mathsf{new}}) := \mathrm{REAL}_{\pi,\mathcal{A}}(C_{\mathbf{c}}, \mathbf{I}, \mathbf{acc}^{\mathsf{old}}),$$

where $\mathbf{acc}^{\mathsf{new}} \in \mathcal{D}_{coin}^{n+1}$ is the balance vector after the protocol execution[13] and $\mathbf{out} \in (\mathcal{D}_{out} \cup \{\mathit{setupFail}, \mathit{abort}\})^n$ is the vector of parties' outputs. The meaning of the output *setupFail* is to signal that the setup phase of the protocol did not complete successfully (which might for example include the situation when malicious parties do not make their initial deposits or when parties disagree on the contract to be executed, etc.). The meaning of the output *abort* is to signal that the setup phase was successful but the contract execution did not complete successfully (this might happen for example if a malicious party does not provide its input or the operator stops communicating with the parties, etc.). In case all parties are honest, for brevity we write only $\mathrm{REAL}_{\pi}(C_{\mathbf{c}}, \mathbf{I}, \mathbf{acc}^{\mathsf{old}})$.
Security definitions We first define the security property called *operator balance security* which, intuitively, says that in case the operator behaves honestly, he cannot lose money.

Definition 1 (Operator balance security). A protocol π run by parties from $\mathcal{P}^+$ satisfies the *operator balance security* property if for every (n, m)-contract $C_{\mathbf{c}}$, for every adversary $\mathcal{A}$ corrupting only parties from $\mathcal{P}$ (i.e., the operator must be honest), for every $\mathbf{I} \in \mathcal{D}_{in}^{n \times m}$ and every $\mathbf{acc}^{\mathsf{old}} \in \mathcal{D}_{acc}(\mathbf{c})$, the output of the protocol execution $(\mathbf{out}, \mathbf{acc}^{\mathsf{new}}) := \mathrm{REAL}_{\pi,\mathcal{A}}(C_{\mathbf{c}}, \mathbf{I}, \mathbf{acc}^{\mathsf{old}})$ is such that $\mathbf{acc}^{\mathsf{new}}[n+1] \geq \mathbf{acc}^{\mathsf{old}}[n+1]$.

Intuitively, correctness states that in case all parties behave honestly (including the operator), every party $P_i \in \mathcal{P}$ outputs the correct result and earns the amount of coins she is supposed to get according to the correct contract execution.

Definition 2 (Correctness). Protocol π run by parties $\mathcal{P}^+$ satisfies the *correctness* property if for every (n, m)-contract $C_{\mathbf{c}}$, every $\mathbf{I} \in \mathcal{D}_{in}^{n \times m}$ and every $\mathbf{acc}^{\mathsf{old}} \in \mathcal{D}_{acc}(\mathbf{c})$, the protocol output $(\mathbf{out}, \mathbf{acc}^{\mathsf{new}}) := \mathrm{REAL}_{\pi}(C_{\mathbf{c}}, \mathbf{I}, \mathbf{acc}^{\mathsf{old}})$ is such that $\forall i \in \mathcal{P}$:

$$\mathbf{out}[i] = \mathit{out} \;\text{ and }\; \mathbf{acc}^{\mathsf{new}}[i] = \mathbf{acc}^{\mathsf{old}}[i] - \mathbf{c}[i] + \mathbf{d}[i],$$

where $(\mathit{out}, \mathbf{d}) := \mathrm{Eval}(C_{\mathbf{c}}, \mathbf{I})$ (recall that $\mathbf{c}$ is the vector of deposits made by parties $\mathcal{P}$ as per contract $C_{\mathbf{c}}$).

Finally, we define the fairness property which, on high level, guarantees that if at least one party $P_i \in \mathcal{P}$ is honest, then (i) either the protocol correctly completes an execution of the contract or (ii) all honest parties output *setupFail* and stay financially neutral or (iii) all honest parties output *abort*, stay financially neutral, and at least one corrupt party must have been punished.

[13] Recall that we consider only standalone security definition; thus the account values cannot be changed by other processes.

Definition 3 (Fairness)**.** A protocol π run by parties from $\mathcal{P}^+$ satisfies the *fairness* property if for every (n,m)-contract $C_{\mathbf{c}}$, for every adversary $\mathcal{A}$ corrupting parties from $\mathcal{P}^+$ such that at least one party from $\mathcal{P}$ is honest, for every $\mathbf{I} \in \mathcal{D}_{in}^{n\times m}$ and every $\mathbf{acc}^{\mathsf{old}} \in \mathcal{D}_{acc}(\mathbf{c})$, the output of the protocol execution $(\mathbf{out}, \mathbf{acc}^{\mathsf{new}}) := \mathrm{REAL}_{\pi,\mathcal{A}}(C_{\mathbf{c}}, \mathbf{I}, \mathbf{acc}^{\mathsf{old}})$ is such that one of the following statements must be true (below $\mathcal{H}^+$ denotes the set of honest parties and $\mathcal{H}$ the set of honest parties excluding the operator):

(i) $\exists \mathbf{I}^* \in \mathcal{D}_{in}^{n\times m}$ such that $\forall i \in \mathcal{H}$ the following holds:

- $\mathbf{I}[i][\cdot] = \mathbf{I}^*[i][\cdot]$,
- $\mathbf{out}[i] = out$ and $\mathbf{acc}^{\mathsf{new}}[i] \geq \mathbf{acc}^{\mathsf{old}}[i] - \mathbf{c}[i] + \mathbf{d}[i]$, where $(out, \mathbf{d}) := \mathrm{Eval}(C_{\mathbf{c}}, \mathbf{I}^*)$.

(ii) $\forall i \in \mathcal{H}$: $\mathbf{out}[i] = setupFail$, $\mathbf{acc}^{\mathsf{new}}[i] \geq \mathbf{acc}^{\mathsf{old}}[i]$.

(iii) $\forall i \in \mathcal{H}$: $\mathbf{out}[i] = abort$, $\mathbf{acc}^{\mathsf{new}}[i] \geq \mathbf{acc}^{\mathsf{old}}[i]$ and

$$\sum_{\ell \in \mathcal{P}^+ \setminus \mathcal{H}^+} \mathbf{acc}^{\mathsf{new}}[\ell] < \sum_{\ell \in \mathcal{P}^+ \setminus \mathcal{H}^+} \mathbf{acc}^{\mathsf{old}}[\ell]. \tag{1}$$

Note that if the statement (i) holds, then the protocol terminated in the *finalize* phase. Hence, the only thing the adversary $\mathcal{A}$ might have done was changing the inputs of the *malicious* parties and/or spending their coins. In other words, an honest party's output is the same as it would be if no party was corrupt but inputs of parties from $\mathcal{P} \setminus \mathcal{H}$ can be different.

C Proof of Theorem 1

We now formally state and proof the main theorem of our work which has been informally explained in Sec. 7.

Theorem 1 (Formal statement)**.** *Assuming existence of a signature scheme* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *that is existentially unforgeable under chosen message attack, a trusted execution environment emulating the TEE ideal functionality (see Sec. 5.2) and a blockchain emulating the* BC *ideal functionality (see Sec. 5.2), the protocol* $\pi_{\text{FastKitten}}$ *as defined in Section 5 satisfies correctness, fairness and operator balance security property.*

For the purpose of the security analysis, we do not consider the fees of posting transaction on the blockchain and the value of the challenge and response transactions μ (see Sec. 5 for the definition of the mentioned transactions).

Proof of operator balance security

We distinguish the following cases when the operator is honest:

Case A: If during the *setup phase*, the participants do not agree on a contract C, then the operator does not make its deposit tx_Q and hence does not lose any coins.

Case B: If the parties agree on the contract during the *setup phase*, the operator makes its deposit but at least one of the parties does not make a deposit, then the enclave outputs a transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, setupFail)$ which the operator can publish on the blockchain. By definition of this transaction, the operator receives its entire deposit back.

Case C: Consider now the situation that the *setup phase* successfully completes. If there is a party P_i which at round $j \in [m]$ of the *round computation* phase does not provide correctly signed input, even after the challenge-response case, then the enclave will then output the transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, abort)$ that returns the deposit back to the operator.

Case D: If the contract execution successfully terminates, the enclave outputs the transaction $\mathsf{tx}_{\mathsf{out}}([n], \mathbf{d}, out_C)$ that returns the entire deposit back to the operator.

In remains to discuss that the output transaction posed by the operator in cases B,C and D is valid when published on the blockchain (i.e. all the input transactions are unspent transactions). The only other transaction that is spending the deposit transactions is the penalty transaction tx_{p} which is valid only after round $\tau_{\mathsf{final}} = \tau + (3 + 2m)(\delta + 5)$, where τ is the starting round, δ upper bounds the blockchain delay, k is the confirmation constant and m is the maximal amount of execution rounds of the contract C. Since the setup phase takes at most $3(\delta + k)$ rounds and each execution of the contract takes at most $2(\delta + k)$ rounds, an honest operator has enough time to post the output transaction to the blockchain.

Proof of correctness

For this security property we consider the scenario when all parties from $\mathcal{P}$ and the operator are honest. The protocol starts with the *Setup phase*, where the parties from $\mathcal{P}$ agree to the contract C, followed by the operator's deposit tx_Q. Once operator's deposit is confirmed on the blockchain, then all parties from $\mathcal{P}$ make their deposit. Next, the protocol proceeds to the *round computation phase.* For every $P_i \in \mathcal{P}$ and every round $j \in [m]$ the following holds: (1) P_i sends input $(in_i,\ s_i)$ to operator P_{n+1}, who (2) confirms that the input $(in_i,\ s_i)$ is correctly signed, i.e. $\mathsf{Vrfy}(pk_i; in_i, s_i) = 1$. The operator loads the input vector $\vec{in}$ into the enclave and which runs the contract C as $(out_C, state', \mathbf{d}) := C(state, \vec{in})$. By our assumption that the contract *contract* is an (n, m)-contract, we know that after at most m rounds the output of the contract execution is $(out_C, \perp, \mathbf{d})$ which signal the final round. The enclave outputs transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{P}, \mathbf{d}, out_C)$, which sends $\mathbf{d}[i]$ coins back to party P_i. Hence, for every $i \in \mathcal{P}$ it holds that $\mathbf{acc}^{\mathsf{new}}[i] = \mathbf{acc}^{\mathsf{old}}[i] - \mathbf{c}[i] + \mathbf{d}[i]$.

Proof of Fairness

Let us first focus on the Setup phase of the protocol $\pi_{\text{FastKitten}}$ and show that if it does not complete successfully, then all honest parties output "*setupFail*" and stay financially neutral.

Lemma 2. *If there exists an honest party P_i such that $\mathbf{out}[i] = setupFail$, then the statement (ii) of the fairness property holds.*

Proof. According to the protocol description, there are three cases when an honest party P_i can output *setupFail*: (i) if P_i did not receive a valid tuple $(\mathsf{init}, \mathsf{tx}_{\mathsf{p}})$ from the operator. In that case the party does not post the transaction tx_i; (ii) if a output transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, setupFail)$ is posted to the blockchain before τ_{final}. In that case, by definition of the function Pdep, at least one party P_{i^*} did not post the transaction tx_{i^*}; and (iii) if P_i is posting the penalty transaction tx_{p} in round τ_{final} and at least one transaction tx_{i^*} was not posted on the blockchain before round τ_{setup}. Hence, we proved that if an honest party P_i outputs "*setupFail*", then there exists $i^* \in \mathcal{P}$ such that tx_{i^*} was not published on the blockchain before round τ_{setup}. Let us now show that the opposite implication holds as well.

An honest party P_i does not post the transaction tx_i to the blockchain only if she does not receive a valid tuple $(\mathsf{init}, \mathsf{tx}_{\mathsf{p}})$. In this case P_i outputs *setupFail.* If P_i posted the transaction tx_i on the blockchain but tx_{i^*} was not posted for some $i^* \neq i$, then the function Pdep (if executed) must return a transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, setupFail)$. If such this output transaction is posted on the blockchain before round τ_{final}, then P_i outputs *setupFail.* If no such transaction is posted on the blockchain, then in round τ_{final} party P_i posts the penalty transaction tx_{p} and return *setupFail.*

We can conclude that if one honest party outputs "*setupFail*", then all honest parties output "*setupFail*". It remains to show that if an honest P_i outputs "*setupFail*", then she never loses coins. Consider again the three possible cases in which an honest P_i outputs "*setupFail*". In the case (i) P_i never makes any deposit and hence cannot lose coins; in cases (ii) and (iii) P_i gets $\mathbf{c}[i]$ coins back by definition of the output, resp. penalty, transaction.

□

The above lemma show that fairness holds in case the Setup phase fails. As a next step we show the more interesting case; namely, that fairness holds also if the Setup phase completes. The lemma below discusses the case when an honest party outputs "*abort*".

Lemma 3. *If there exists an honest party P_i such that $\mathbf{out}[i] = abort$, then the statement (iii) of the fairness property holds.*

Proof. According to the protocol description, there are two cases when an honest party P_i outputs "*abort*": (i) a transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, abort)$ is published on the blockchain before round τ_{final} or (ii) all transactions tx_i were posted to the blockchain before round τ_{setup} and the transaction tx_{p} was posted on the blockchain latest in round τ_{final}. In a similar way as in the proof of Lemma 2, we can prove that the opposite implication holds as well and hence, if one honest party outputs "*abort*", then so do all honest parties. It remains to discuss that no honest party loses coins and that malicious parties are punished.

Let us first consider the case (i). By definition of the FastKitten enclave program, the enclave outputs a transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{d}, abort)$ if and only if the operator executes the function $\mathsf{errorProof}$ and parties from the set $\mathcal{P} \setminus \mathcal{J}$, which must not be an empty set, did not submit a valid input for the next round even if it was challenged. Since an honest party always provides input for the next phase, $\mathcal{H} \subseteq \mathcal{J}$. By definition of the output transaction, all parties from $\mathcal{J}$ and the operator get their deposit back and the parties from $\mathcal{P} \setminus \mathcal{J}$ do not get anything, i.e. $\forall \ell \in \mathcal{P} \setminus \mathcal{J}$ it holds that $\mathbf{acc}^{\mathsf{new}}[\ell] = \mathbf{acc}^{\mathsf{old}}[\ell] - \mathbf{c}[\ell]$. Since $\mathbf{c}[\ell] > 0$ and $\mathcal{P} \setminus \mathcal{J} \neq \emptyset$, at least one malicious party lost coins. In addition, we know that no malicious party earned coins. Hence, the inequality (1) holds.

In case (ii) the penalty transaction is posted on the blockchain which mean that every $P_i \in \mathcal{P}$ gets $\mathbf{c}[i]$ coins back, i.e. $\mathbf{acc}^{\mathsf{new}}[i] = \mathbf{acc}^{\mathsf{old}}[i]$ and the operator loses all the deposited coins, i.e. $\mathbf{acc}^{\mathsf{new}}[i] = \mathbf{acc}^{\mathsf{old}}[i] - \sum_{i \in [n]} \mathbf{c}[i]$. The operator balance property implies that the operator must be malicious. Since malicious operator lost coins and no other malicious party earned any coins, the inequality (1) holds.

□

What remains to show it that the fairness property holds when the execution of the contract successfully completes.

Lemma 4. *If there exists an honest party P_i such that $\mathbf{out}[i] \notin \{abort, setupFail\}$, then statement (i) of the fairness property holds.*

Proof. According to the protocol description, an honest party P_i outputs $\mathbf{out}[i] \notin \{abort, setupFail\}$ if only if a transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{d}, \mathbf{out}[i])$ is posted on the blockchain before round τ_{final}. From Lemma 2 and Lemma 3 we know that for every honest party P_ℓ it must hold that $\mathbf{out}[\ell] \notin \{abort, setupFail\}$. Since only one transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{d}, out)$ can be posted on the blockchain, all honest parties output the same value $out \in \mathcal{D}_{out}$. Hence, we proved that honest parties reach consensus on the output value.

By definition of the function round of the FastKitten enclave program, the enclave outputs a transaction $\mathsf{tx}_{out}(\mathcal{J}, \mathbf{d}, out)$ if only if the loaded contract terminates with $\mathbf{d}$ as the final distribution and out as the final output before round τ_{final}. This, in particular, implies that every party P_i provided a signed input $in_{i,j}$ in every round j and the operator loaded all of them into the enclave, i.e. $\mathbf{I}^*[i][j] := in_{i,j}$. Unforgeability of the signature scheme guarantees that for every honest party P_i and every round j it holds that $in_{i,j} = \mathbf{I}[i][j]$, i.e. honest parties' inputs cannot have been modified. Since the function at least one party $P_i \in \mathcal{P}$ is honest, we know that the executed contract inside the enclave is indeed $C_\mathbf{c}$ (if the operator load s a different contract inside the enclave, the verification of the quote would fail in which case an honest party outputs "*setupFail*".). □

D Implementation of Provably Fair Poker

```
// Pre-Flop
Common cards:
Your hand:    [9♠] [T♦]
Player 1 bets 1000
Pot: 1000  Your bet: 0  Stack: 10000
Call/Fold/Bet (min 1000) >  1000

// Flop
Common cards: [3♠] [J♥] [9♥]
Your hand:    [9♠] [T♦]  (Pair)
Player 1 bets 1000
Pot: 3000  Your bet: 1000  Stack: 9000
Call/Fold/Bet (min 1000) >  1000

// River
Common cards: [3♠] [J♥] [9♥] [Q♥]
Your hand:    [9♠] [T♦]  (Pair)
Player 1 bets 1000
Pot: 5000  Your bet: 2000  Stack: 8000
Call/Fold/Bet (min 1000) >  1000

// Turn
Common cards: [3♠] [J♥] [9♥] [Q♥] [7♦]
Your hand:    [9♠] [T♦]  (Pair)
Player 1 bets 1000
Pot: 7000  Your bet: 3000  Stack: 7000
Call/Fold/Bet (min 1000) >  1000
Player 2 wins 8000 with a Pair [9♠] [T♦]
```

Figure 5: Example match of Texas Hold'em Poker in FastKitten with two players placing bets in real time.

For our second use case we implemented a contract that allows participants to play provably fair online poker. We based the Python contract on a poker library for Texas Hold'em [65] and modified it by adding a player chip stack and the possibility of betting a number of chips into the pot each round. Participants first have to buy into the game with an equal amount of bitcoins, which will be converted to a 10K chip stack for each player sitting at the table (in our protocol, this happens during the *Initialize* step of the setup phase).

After the setup phase and before the first round each player is assigned a seat number by the enclave. The enclave additionally chooses one of the players at random and assigns the dealer button to that party. Then each player is dealt a hand by the enclave and every player can decide whether or not to place a bet, starting from the player sitting next to the dealer button (for brevity we omitted the necessity of placing blinds). After the flop, river, and turn have been dealt by the enclave players have a final chance of placing bets. If more than two players remain in the game after the final bets the enclave reveals the winner and distributes the chips in the current pot and schedules the next round. Further rounds are scheduled until either the maximum number of matches have been played or only one player remains with all the chips at the table. An example player view of a round in which two players placed bets is depicted in Listing 5. In our evaluation we did not include a thinking time per player: we automated the interaction using bots that placed bets instantaneously based on a win strategy instead of using human players.

E Transaction Transcripts

The following Listings show sample penalty, challenge, response, and output transactions, as shown by the command `decoderawtransaction` of Bitcoin Core 0.16.1. Those transactions were generated for the test Bitcoin network (testnet); the main network (mainnet) would use very similar transactions (of the same length). Only the relevant parts are shown.

```
{
  "size": 504,
  "locktime": 1446631,
  "vin": [
  {
      "txid": "84e5253e199d3b4a100a385381be90cf2ff093608dafd8042c6eface3c16f453",
      "vout": 11,
      "scriptSig": {...},
      "txinwitness": [...],
      "sequence": 4294967294
    }
  ],
  "vout": [
  {
      "value": 0.00999950,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_HASH160 3210972ae2b0dd7a0f908d0561f774e05b3bed46 OP_EQUAL",
        "hex": "a9143210972ae2b0dd7a0f908d0561f774e05b3bed4687",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2MwowebxQDF1s37WELjUAnGYQbQ3xkb1f4c" ] // Player 0
      }
    }, {
      "value": 0.00999950,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_HASH160 555c9781aebcaae0ccb7993c1f1549784c04fc9d OP_EQUAL",
        "hex": "a914555c9781aebcaae0ccb7993c1f1549784c04fc9d87",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2N12aLyvDcmidHkKQavVLjJvhQA27fLEe7v" ] // Player 1
      }
    }, ... {
      "value": 0.00999950,
      "n": 9,
      "scriptPubKey": {
        "asm": "OP_HASH160 5a463456cf6a22e899b5aa2f55cce5af1b6ef89d OP_EQUAL",
        "hex": "a9145a463456cf6a22e899b5aa2f55cce5af1b6ef89d87",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2N1UYuSSGzfiKtDQcDMiu9QFNiDZy8MUcgB" ] // Player 9
      }
    }
  ]
}
```

Listing 1: A sample penalty transaction (size: 504 bytes).

```
{
  "size": 293,
  "locktime": 0,
  "vin": [
    {
      "txid": "84e5253e199d3b4a100a385381be90cf2ff093608dafd8042c6eface3c16f453",
      "vout": 11,
      "scriptSig": {...},
      "txinwitness": [...],
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00000100,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_HASH160 3210972ae2b0dd7a0f908d0561f774e05b3bed46 OP_EQUAL",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2MwowebxQDF1s37WELjUAnGYQbQ3xkb1f4c" ] // Player 0
      }
    }, {
      "value": 0.84998953,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_HASH160 54cb7b683ce803cc38a93f4686fc818fdcdc2251 OP_EQUAL",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2MzyaWaYAYKSmn9wL5aP6m6GtVa9xe8ppJe" ]
      }
    }, {
      "value": 0.00000000,
      "n": 2,
      "scriptPubKey": {
        "asm": "OP_RETURN
            436f6d6d6f6e3a203353204a48203948205148203744202048616e643a203953205444",
        "hex": "
            6a23436f6d6d6f6e3a203353204a48203948205148203744202048616e643a203953205444
            ",
        "type": "nulldata"
      }
    }
  ]
}
```

Listing 2: A sample challenge transaction (size: 293 bytes).

```
{
  "size": 266,
  "locktime": 0,
  "vin": [
    {
      "txid": "85248aafd413c612a9baf2827862695802748374f7c8635edeb7ade4f3b9d2c1",
      "vout": 0,
      "scriptSig": {...},
      "txinwitness": [...],
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00000100,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_HASH160 2d2d7a1de4b8f57804c3101f73d422ede2feedc2 OP_EQUAL",
        "hex": "a9142d2d7a1de4b8f57804c3101f73d422ede2feedc287",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2MwN6siRR7DthC8jgyoAg9zjrQxQUmM5ynw" ]
      }
    }, {
      "value": 1.04999466,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_HASH160 2acecdf3b5a432057412c898361a0433c5539713 OP_EQUAL",
        "hex": "a9142acecdf3b5a432057412c898361a0433c553971387",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2Mw9a7NNtj9wd4ozky47LNLD28MJ753y5qd" ]
      }
    }, {
      "value": 0.00000000,
      "n": 2,
      "scriptPubKey": {
        "asm": "OP_RETURN 4265742031303030",
        "hex": "6a084265742031303030",
        "type": "nulldata"
      }
    }
  ]
}
```

Listing 3: A sample response transaction (size: 266 bytes).

```
{
  "size": 1986,
  "locktime": 0,
  "vin": [
    { // Deposit by operator
      "txid": "84e5253e199d3b4a100a385381be90cf2ff093608dafd8042c6eface3c16f453",
      "vout": 2,
      "scriptSig": { ... },
      "txinwitness": [ ... ],
      "sequence": 4294967295
    }, { // Deposit by player 0
      "txid": "84e5253e199d3b4a100a385381be90cf2ff093608dafd8042c6eface3c16f453",
      "vout": 0,
      "scriptSig": { ... },
      "txinwitness": [ ... ],
      "sequence": 4294967295
    }, { // Deposit by player 1
      "txid": "84e5253e199d3b4a100a385381be90cf2ff093608dafd8042c6eface3c16f453",
      "vout": 3,
      "scriptSig": { ... },
      "txinwitness": [ ... ],
      "sequence": 4294967295
    }, ... { // Deposit by player 9
      "txid": "84e5253e199d3b4a100a385381be90cf2ff093608dafd8042c6eface3c16f453",
      "vout": 11,
      "scriptSig": { ... },
      "txinwitness": [ ... ],
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.09999000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_HASH160 54cb7b683ce803cc38a93f4686fc818fdcdc2251 OP_EQUAL",
        "hex": "a91454cb7b683ce803cc38a93f4686fc818fdcdc225187",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2MzyaWaYAYKSmn9wL5aP6m6GtVa9xe8ppJe" ] // Operator
      }
    }, {
      "value": 0.09999000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_HASH160 850898c171a3de1d180120f2f9d30f3edada6933 OP_EQUAL",
        "hex": "a914850898c171a3de1d180120f2f9d30f3edada693387",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [ "2N5NeA79D7YV4gM3JmcWxHWDnYnx1BX6q1S" ] // Player 4
      }
    }, {
      "value": 0.00000000,
      "n": 2,
      "scriptPubKey": {
        "asm": "OP_RETURN 506c61796572203420776696e732e",
        "hex": "6a0e506c61796572203420776696e732e",
        "type": "nulldata"
      }
    }
  ]
}
```

Listing 4: A sample output transaction (size: 1986 bytes).