How To Save Fees in Bitcoin Smart Contracts: a Simple Optimistic Off-chain Protocol

Dario Maddaloni, Riccardo Marchesin and Roberto Zunino

Università degli studi di Trento

Abstract

We consider the execution of smart contracts on Bitcoin. There, every contract step corresponds to appending to the blockchain a new transaction that spends the output representing the old contract state, creating a new one for the updated state. This standard procedure requires the contract participants to pay transaction fees for every execution step.

In this paper, we introduce a protocol that moves most of the execution of a Bitcoin contract off-chain. When all participants follow this protocol, they are able to save on transaction fees. By contrast, in the presence of adversaries, any honest participant is still able to enforce the correct execution of the contract, according to its original semantics.

Keywords

Smart contracts, Bitcoin, off-chain protocols, optimistic protocols

1. Introduction

Moving the computation off-chain to save on fees has become a popular means to scale the blockchain smart contracts processing capabilities, making them more practical. To this aim, a variety of methods has been considered, in particular over account-based blockchain such as Ethereum [1, 2, 3]. In this work, we instead focus on Bitcoin smart contracts.

While limited by a non Turing-complete scripting language, Bitcoin still allows the implementation of a broad class of smart contracts [4, 5]. Some simple contracts can be expressed using a single Pay-to-Script-Hash transaction that encodes a suitable spending condition. However, exploiting multiple transactions it is possible to implement more complex Bitcoin contracts, allowing multiple rounds of interaction among participants [7, 6].

Users who want to deploy a multi-transaction smart contract on Bitcoin can do so with the following standard technique: first, they generate a tree of transactions that describes the possible evolution of the contract, then they sign all the transactions in the tree, and finally they append to the blockchain only the transactions in a single tree path, according to participants' choices. A downside of this approach is that every contract step corresponds to appending a transaction to the blockchain: this is expensive in terms of fees.

In this paper we propose a protocol to move most of the execution off-chain, while still guaranteeing the same contract behaviour. In this protocol participants simulate the contract by exchanging off-chain signatures. In the optimistic case, in which all participants are honest and follow the protocol correctly, they only need to append three transactions to the blockchain. This does not depend on the number of transactions in the original contract.

Our off-chain protocol has a failsafe mechanism that can be triggered whenever someone detects malicious behaviour. This mechanism moves the contract execution back on-chain, safeguarding the contract behaviour from malicious actors. Even in this negative scenario, the steps that were completed off-chain are preserved, reducing the amount of on-chain steps needed to get the contract to completion. When the failsafe is triggered, participants need to pay the fees associated with the remaining contract steps and wait for a few additional time delays. The fees that are saved by the off-chain execution can be

DLT24, 6th Distribute Ledger Technology Workshop, May 14-15 2024, Torino

dariomaddaloni.6@gmail.com (D. Maddaloni); riccardo.marchesin@unitn.it (R. Marchesin); roberto.zunino@unitn.it (R. Zunino)

distributed to the participants when the contract terminates: this serves as an incentive for participants to correctly execute the off-chain protocol.

While our technique is designed to be executed on Bitcoin as-is, it only relies on the fundamentals of the UTxO model, as well as timelocks, a widely adopted primitive in UTxO blockchains, making our approach easily adaptable to other UTxO platforms.

Contributions We summarize our contributions as follows:

- We outline a general technique to transform a on-chain Bitcoin contract into an off-chain contract.
- We showcase our technique through a simple example.
- We discuss the security and efficiency of our protocol.

2. On-chain contracts

We represent contracts C as trees of transactions, as in Figure 1. Transactions highlighted in yellow are those already on the blockchain. We will assume that every participant of the contract (denoted with $\mathsf{A}, \mathsf{B}, \cdots$) spends at least one of their transaction output into the first transaction of the contract. Edges in the tree represent requirements that need to be satisfied in order for a transaction to be redeemed. Such restrictions are either set by the script of the "parent" transaction (for instance asking to sign the redeeming transaction with a specific key, or to reveal the preimage of a given hash), or they can be set within the redeeming transaction itself, like in the case of timelocks, that force the transaction to wait for a certain amount of blocks before it can be appended.

We assume that all the edges represented with solid lines implicitly ask for a signature of every contract participant in addition to the signatures specified along the edges. These extra signatures are crucial to the contract stipulation protocol: we will refer to them as *implicit signatures*. In fact, this requirement ensures that if at least one participant is honest (i.e. wants to abide by the original semantics of the contract), then the others can not deviate from the intended behaviour of the contract, since doing so would require a signature from every participant (including the honest ones).

To stipulate a contract, participants first exchange messages containing every transaction in the tree. Then, they exchange all the implicit signatures on each transaction. Finally, they exchange the signatures for the root transaction T_0 and put it on the blockchain, ending the stipulation phase. We stress that the whole tree must be signed before T_0 is. Not doing so would allow an adversary to put a part of the tree on chain and then prevent the rest of the contract to be executed by refusing to sign the continuation.

Once T_0 is on the blockchain, participants can execute the contract by following a branch of the tree. At each step they have to satisfy the requirements shown on the edges of the tree, and then put the corresponding transaction on chain. Note that, during the execution of the contract, the implicit requirements are already satisfied, since those signatures have already been exchanged during the stipulation phase.

Example: Best-of-3 Bet We illustrate the on-chain execution of a contract through an example. The tree of transactions presented in Figure 2 implements a best-of-three bet between Alice and Bob, on some kind of recurring match between two teams. Before the competition starts, an oracle commits the hashes of six secret messages: three of them denote that the first team won a match, while the other three denote that it lost [1]. After each match the oracle certifies the victory or loss of the team by revealing the suitable secret message. These secrets can then be used by Alice and Bob to update the state of the contract, appending a new transaction to the blockchain. Transactions in the contract are

[1] The 6 messages contain the strings $W_1, W_2, W_3, L_1, L_2, L_3$ and are padded with a nonce for security. Note that the oracle is not a contract participant, and does not need to be aware of the bet between Alice and Bob. Indeed its role could be performed by three distinct oracles, one for each match.

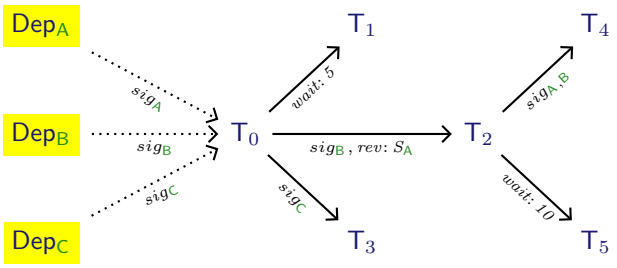


Figure 1: Example of a contract. sig_P labels require an authorization by P (signature on the redeeming transaction). rev: S labels require revealing the secret S whose hash was previously committed. $wait$: t labels require waiting for at least t blocks

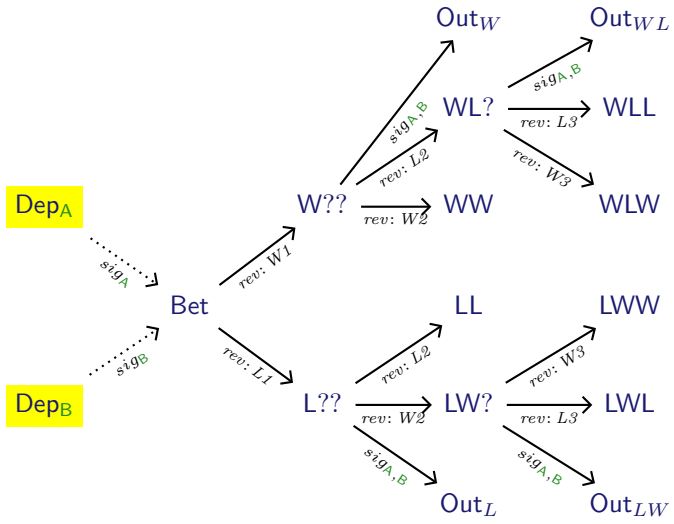


Figure 2: Best-of-3 Bet

named to represent the win-loss record associated to the corresponding state, from the point of view of the first team, meaning that W?? represents the state in which the first team has won once, and two matches still have to be played, while LWL represents the state in which the first team has lost, then won, then lost again. The contract ends when either team has won two out of three matches, or when both Alice and Bob agree to an early payout, terminating the bet before the three matches have been played. The contract has 10 possible terminations, shown in the tree as 10 leaves.

The transactions WLW, WW transfers all the balance of the contract to Alice, who bet on the first team winning. Likewise, the transactions LL, LWL transfer the prize to Bob. The early payout option, that can be taken only if Alice and Bob both agree on it, is performed by nodes Out$_W$ (which gives a bigger share to Alice, since it can be taken after the first win has been revealed); Out$_{WL}$ and Out$_{LW}$ (which split the prize equally), and Out$_L$ (which pays a bigger share to Bob).

It is important to notice that in this contract the closer a leaf is to the root, the more total money it will have available, due to having to pay less in fees. For this reason, the transactions WW, LL, Out$_W$, Out$_L$ can pay back to Alice and Bob a part of the fees that they anticipated. In such a small contract, the difference is only slight, but in deeper contracts, i.e. a best-of-5 bet, they start to become more impactful. Figure 3 shows a possible on-chain execution of the contract.

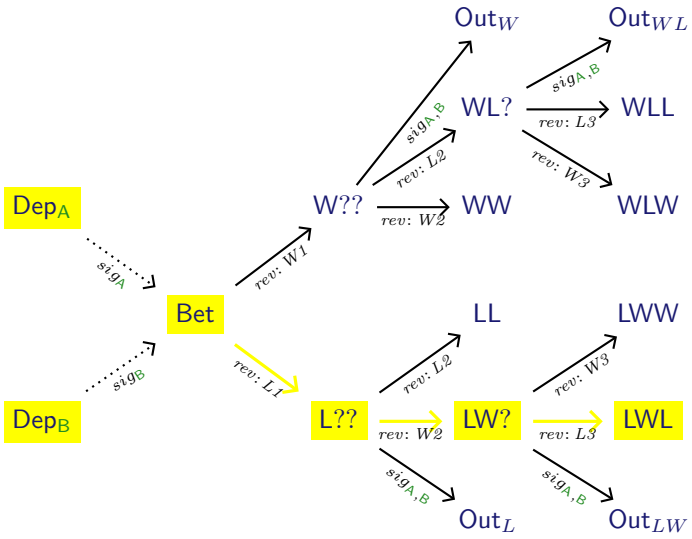


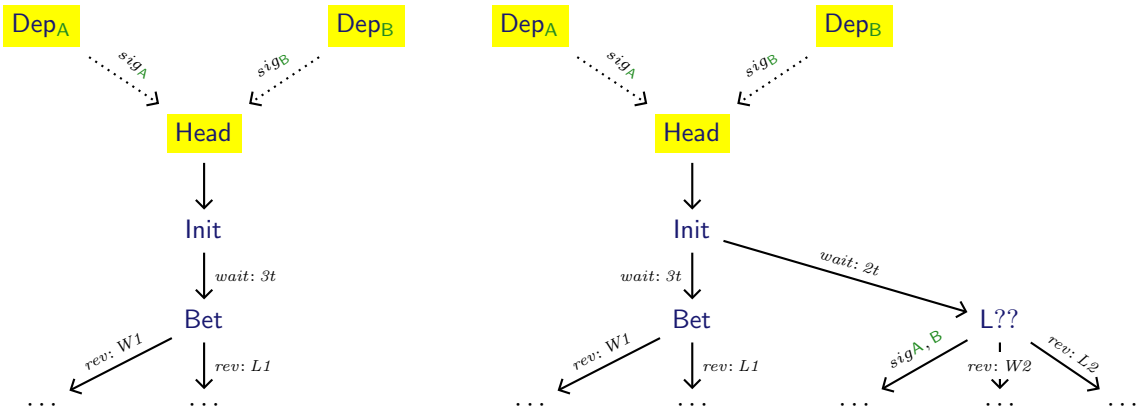
Figure 3: A possible execution path. Every time the oracle de-commits one of the hashes, a participant can exhibit its preimage in the contract, moving to the next state.

3. Off-chain contracts

We will now walk through the steps that are needed to move off-chain the execution of the best-of-three bet. First, Alice and Bob generate a slightly modified copy of the contract (see Figure 4a), where there is a new Head transaction that has as inputs the two inital deposits, an Init transaction (whose input is Head) and a modified Bet transaction, which now features a timelock of $3t$ relative to its input, Init. From there, the modified contract follows the structure of the original one. Again, mirroring the protocol described for the on-chain stipulation of the contract, Alice and Bob send each other the implicit signatures needed for every transaction, leaving for last the ones that unlock the two deposits and enable Head. Once they have all the needed signatures, they append Head to the blockchain, spending their deposits. Notice that at this point the transaction Init is enabled (since they had already exchanged the signatures) but, differently from the on-chain contract execution, Alice and Bob should refrain from appending it to the blockchain right now. Indeed, the Init transaction will only be appended when a participant decides to move the execution back on-chain. If both are honest this will happen only when the contract has reached a leaf.

From now on, our example will follow a potential execution trace of the contract, presented in Figure 3, assuming that the oracle will reveal "L1", "W2", and "L3", and that neither Alice nor Bob will agree to take an early payout. After the oracle reveals "L1", in the original on-chain contract Bob would append L?? to the blockchain moving the state forward. Instead, in the off-chain protocol, the two participants create a *graft*: a copy of the subtree rooted at L??, modifying L?? so that its input is now the Init transaction, and so that it has a relative timelock of $2t$. After grafting this subtree to the off-chain contract (see Figure 4b), Alice and Bob exchange the implicit signatures needed by the transactions in the subtree. If at this point either participant wants to bring the contract execution on-chain, then they can do so by appending the Init transaction, waiting for $2t$ units of time, and then appending L??, carrying on with the on-chain execution from that point onward. Note that, after Init has been appended to the blockchain, an adversary could attempt to "roll back" the contract to a previous state, by appending Bet instead of L??. However, due to the delays enforced by the timelocks, any honest participant is able to prevent this by appending L?? (which unlocks earlier than Bet) as soon as it is enabled.

The next off-chain steps are performed in a similar fashion: Alice and Bob can graft a copy of the subtree of transactions rooted in the one that represents the next state, modifying it so that the root has as input the transaction Init, and so that it now includes a timelock which is crucially smaller than



(a) Start of the off-chain tree (b) After the first step, a subtree is grafted

Figure 4:

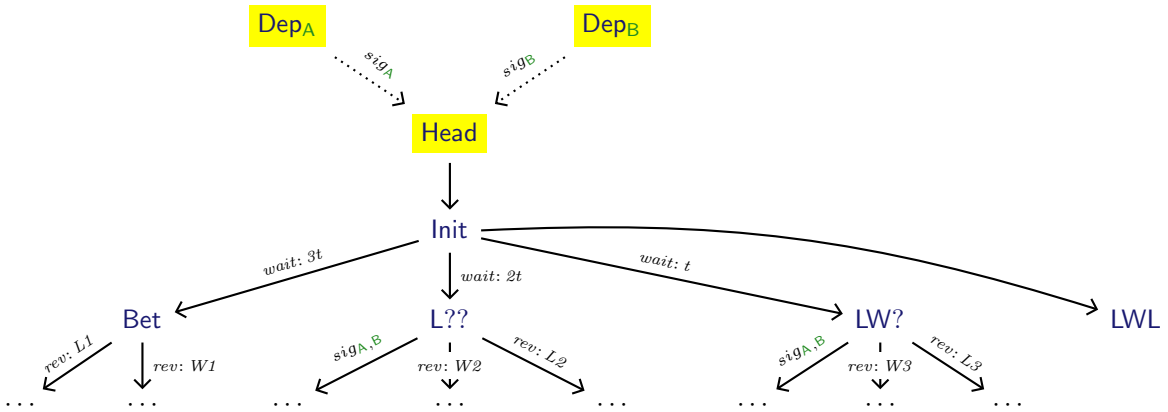


Figure 5: Complete off-chain contract

the ones created in the previous steps. After the graft is created, they exchange the implicit signatures, leaving for last the ones for the subtree root.

Assuming now that Alice and Bob are continuing the off-chain execution, they will wait for the oracle to reveal the next result (in this case "W2"), then graft the subtree rooted at LW?. Finally, after "L3" is revealed, they can graft the subtree consisting only of the transaction LWL, getting the full off-chain tree shown in Figure 5. Now that the contract has no further possible steps, Alice and Bob complete the execution by putting both Init, and LWL on-chain. In this way, the contract is completed with only 3 on-chain transactions (Head, Init, and LWL), when the on-chain protocol required 4 transactions to for the same execution path. Consequently, the participants have saved one fee, which can be redistributed by LWL.

The protocol Moving away from the example, we present the general off-chain execution protocol for an arbitrary contract. Given an on-chain contract tree, participants follow these steps:

1. Create the transactions Head (which redeems one deposit for each contract participant), Init (which redeems Head), and create a modified copy of the contract, such that the root of the contract can now redeem Init after waiting for a number of time units equal to the depth of the original contract tree. Every one of these transactions will require a signature from every contract participant in order to be redeemed (much like the implicit signatures of the on-chain contract).

2. Exchange the implicit signatures for every transaction.

3. Spend the deposits by signing Head and appending it on-chain.
4. Repeat the following until a leaf is reached. If during any of the following steps any participant does not cooperate, then append Init. As soon as Init is appended (by any participant) break out of the loop.
    a) Choose a possible branch of the contract, satisfy the requirements that were set by the edge (i.e. waiting, revealing hashes, sending signatures).
    b) Create a copy of the subtree rooted in the chosen state, and graft it (off-chain) to the Init transaction, with a timelock that enforces waiting for a number of time units equal to the subtree depth.
    c) Exchange all the implicit signatures for the transaction in the subtree. The signatures for the root are exchanged last.
5. If Init has not been appended, then put it on-chain.
6. Append the root of the latest graft as soon as its timelock expires. If the graft is not a leaf, continue the execution on-chain, redeeming the transactions of the subtree.

We remark that the signatures generated in Item 4c are indeed needed, and the previously generated signatures cannot be reused. This is because the transactions in the grafts have different input fields from the ones in the original tree, and signatures must be calculated on the whole transaction. Like in the on-chain contract protocol, we emphasize that signing the whole graft before its root is essential to avoid stalling attacks. For this reason we sign the transactions step-by-step while the contract is being executed.

Threats Adversaries wishing to disrupt the protocol may attempt to spend contract transactions on the blockchain in order to diverge from the contract behaviour, or they may attempt to prevent some contract step to be performed.

In the presence of at least one honest participant, spending a contract transaction in a completely unintended way is impossible, since every edge of the tree requires a signature from every participant. Such attacks are therefore limited to appending a transaction among those that are signed during the protocol.

Instead, attempts to stall the contract execution are more subtle, since an adversary may simply fail to cooperate with the honest participants, as hinted in the 4th phase of the protocol. For instance, malicious participants might:

- Disagree on the next branch of the contract at item 4a (or agree, but refuse to send the eventual signatures and hash required by the edges of the contract).
- Refuse to send the signatures needed at item 4c within a reasonable time, stalling the execution of the contract.
- Put Init on-chain prematurely, while the next contract step is still being negotiated/executed.

To thwart such attacks, honest participants must vigilate on the contract execution, and react accordingly to recover the correct execution of the contract. Upon detecting one of the above malicious behaviours, an honest participant must move the execution on-chain by appending the Init transaction (if not already on-chain), and then by appending the *latest* graft. This ensures that the original contract behaviour is preserved.

3.1. Assessment

We make a comparison between our protocol for off-chain execution and the standard on-chain contract execution. The main benefits of our new protocol are the following:

- Reduced fees in the best case. If the contract is brought to completion through off-chain steps, then only three transactions are appended on the blockchain, independently from the size of the original contract. This significantly reduces the cost for deep contracts. Moreover, even if the contract is only partially executed off-chain, two off-chain steps are enough to bring the execution cost on par with the original on-chain contract. Any additional off-chain step further reduces the cost.

- No state rollbacks. Under the assumption that at least one contract participant is honest, we have that if an off-chain step has been completed, no adversary will be able to put on-chain a transaction related to any previous state. Indeed, after Init is appended on-chain, the next transaction to be appended will be the root of a graft. Since the latest grafts are enabled first, an honest participant will always be able to redeem Init with the latest graft before an earlier one can be appended.

- No additional waiting in the best case. With every off-chain step, the timelock on the next graft decreases, reaching 0 when a leaf is reached.

On the other hand, the main drawbacks of off-chain execution are the following:

- Slightly increased fees in the worst case. If the off-chain execution is immediately stopped, then the Init transaction is put on-chain right after it is enabled, and the only available continuation consists of the transaction that was the root of the original on-chain contract. So, we have an overhead of two additional transactions (Head and Init).

- Redeeming Init can require waiting. Except in the best case (where the contract is completed off-chain), the failsafe mechanism requires the grafted subtrees to have timelocks. This means that whenever a participant is forced to move on-chain due to an attack attempt, they will have to wait before they can resume the execution on-chain. Like with fees, this is more harmful in the case of an early attack, where the timelocks are larger.

- Increased number of signatures and messages. At every off-chain contract step the participants need to exchange signatures for every transaction in the grafted subtree. The worst case scenario happens when the original contract tree is a single chain with n nodes: this leads to each participant sending $O(n^2)$ additional messages.

When considering fees, our protocol provides significant benefits in almost every scenario. The main drawback seems to be due to timelocks, since a malicious participant can delay the completion of a contract by an amount of time that grows linearly with the depth of the contract tree. The increased number of signatures does not seem too detrimental, especially when considering that for balanced contract trees the overhead is only of $O(n)$ messages.

We also note that attacks are discouraged by the fact that the fees that are saved due to a successful off-chain execution are then redistributed to the contract participants.

Limitations We remark a few limitations of our approach. First, participants must always be live and monitor the blockchain for malicious behaviours, reacting to them in a timely fashion. Doing so assumes that adversaries are not able to perform DoS attacks that can stall honest participants for a significant amount of time (i.e. longer than the failsafe timelocks). For this, we assume that in our timelocks one unit of time is long enough to ensure that honest participants have plenty of time to act, even in presence of DoS attacks. Note that this assumption is widespread, since it is applied to all the time-based blockchain protocols, such as hashed time locked contracts [6].

Our approach computes the timelocks according to the height of the original contract tree, which must be finite and statically known. This assumption is shared with the on-chain execution protocol. This is inevitable when dealing with the Bitcoin platform, due to the limited expressiveness of its scripting language.

Finally, in our protocol, all the contract fees must be provided in advance, and locked within the Head transaction. Additionally, we must specify in advance how much each transaction in the tree will pay as fee: this happens at signing time, much sooner than when fees are actually paid. Only after the contract is terminated the locked but unspent fees are refunded to participants. This problem is also shared by on-chain contract protocols. In the off-chain protocol the problem is mitigated by the fact that fees can be adjusted, following the market fluctuations, when creating the new grafts. However, once the execution is moved on-chain fees are again locked.

4. Conclusions

We presented an optimistic protocol for the off-chain execution of Bitcoin smart contracts. Our protocol allows user to save on transaction fees.

The safety of our protocol is based on the fact that honest participants hold some transactions (the latest graft) that can be put on chain whenever needed in order to commit the latest state to the blockchain. The efficiency of our protocol follows from participants not having to put these transaction on chain until the very end of the contract (if all participants are honest).

This mechanism of floating transactions is reminiscent of the one exploited by the Lightning Network Protocol [8]. While this is a widely adopted and studied protocol, it only enables a limited subset of contracts (mainly focusing on micropayment channels). By contrast, our protocol can be applied to a generic contract tree. Other techniques to efficiently execute Bitcoin contracts rely on extending the Bitcoin protocol to add a new signature type [9]. Finally, some approaches trade off a more complex infrastructure in exchange more flexible smart contract: for instance [10] employs an external trusted execution environment to certify the contract execution, while [11] creates a new layer 2 blockchain that uses Bitcoin as part of its consensus protocol. By contrast, our protocol can be executed on stock Bitcoin without requiring any modification to the Bitcoin protocol, enlarging the trusted computing base, or overlaying a complex blockchain infrastructure.

References

[1] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, E. W. Felten, Arbitrum: Scalable, private smart contracts, in: USENIX Security Symposium, 2018. URL: http://stevengoldfeder.com/papers/Arbitrum-USENIX.pdf.

[2] C. Li, B. Palanisamy, R. Xu, Scalable and privacy-preserving design of on/off-chain smart contracts, in: 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), 2019, pp. 7–12. doi:10.1109/ICDEW.2019.00-43.

[3] A. De Salve, L. Franceschi, A. Lisi, P. Mori, L. Ricci, L2DART: A trust management system integrating blockchain and off-chain computation, ACM Trans. Internet Technol. 23 (2023). URL: https://doi.org/10.1145/3561386. doi:10.1145/3561386.

[4] M. Bartoletti, T. Cimoli, R. Zunino, Fun with Bitcoin smart contracts, in: ISoLA, 2018, pp. 432–449. doi:10.1007/978-3-030-03427-6_32.

[5] M. Bartoletti, R. Zunino, BitML: a calculus for Bitcoin smart contracts, in: ACM CCS, 2018. doi:10.1145/3243734.3243795.

[6] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, R. Zunino, SoK: unraveling Bitcoin smart contracts, in: POST, volume 10804 of *LNCS*, Springer, 2018, pp. 217–242. doi:10.1007/978-3-319-89722-6.

[7] M. Andrychowicz, S. Dziembowski, D. Malinowski, L. Mazurek, Secure multiparty computations on Bitcoin, in: IEEE S & P, 2014, pp. 443–458. doi:10.1109/SP.2014.35, first appeared on Cryptology ePrint Archive, http://eprint.iacr.org/2013/784.

[8] J. Poon, T. Dryja, The Bitcoin Lightning Network: Scalable off-chain instant payments, 2015. URL: https://lightning.network/lightning-network-paper.pdf.

[9] C. Decker, R. Russell, eltoo : A simple layer 2 protocol for Bitcoin, 2018. URL: http://diyhpl.us/~bryan/papers2/bitcoin/eltoo.pdf.

[10] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, A.-R. Sadeghi, FASTKITTEN: practical smart contracts on bitcoin, in: Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19, USENIX Association, USA, 2019, p. 801–818.

[11] sBTC working group, Stacks: a Bitcoin Layer for Smart Contracts, 2023. URL: https://stx.is/nakamoto.