Opportunistic Algorithmic Double-Spending:
How I Learned to Stop Worrying and Love the Fork

Nicholas Stifter[1,2(✉)], Aljosha Judmayer[1,2], Philipp Schindler[1,2], and Edgar Weippl[1,2]

[1] University of Vienna, Vienna, Austria
[2] SBA Research, Vienna, Austria
{nstifter,ajudmayer,pschindler,eweippl}@sba-research.org

Abstract. In this paper, we outline a novel form of attack we refer to as Opportunistic Algorithmic Double-Spending (*OpAl*). *OpAl* attacks *avoid equivocation*, i.e., do not require conflicting transactions, and are *carried out automatically* in case of a fork. Algorithmic double-spending is facilitated through transaction semantics that dynamically depend on the context and ledger state at the time of execution. Hence, *OpAl* evades common double-spending detection mechanisms and can *opportunistically* leverage forks, even if the malicious sender themselves is not responsible for, or even actively aware of, any fork. Forkable ledger designs with expressive transaction semantics, especially stateful EVM-based smart contract platforms such as Ethereum, are particularly vulnerable. Hereby, the cost of modifying a regular transaction to opportunistically perform an *OpAl* attack is low enough to consider it a viable default strategy. While Bitcoin's stateless UTXO model, or Cardano's EUTXO model, appear more robust against *OpAl*, we nevertheless demonstrate scenarios where transactions are *semantically malleable* and thus vulnerable. To determine whether *OpAl*-like semantics can be observed in practice, we analyze the execution traces of 922 562 transactions on the Ethereum blockchain. Hereby, we are able to identify transactions, which may be associated with frontrunning and MEV bots, that exhibit some of the design patterns also employed as part of the herein presented attack.

Keywords: Double-spending attack · Blockchain · Cryptocurrency · Fork

1 Introduction

Double-spending attacks in cryptocurrencies are primarily considered in two general categories. In the first category, an adversary is either themselves capable, or is able to coerce others, to carry out an attack that undermines the expected security guarantees of the underlying consensus protocol [54]. Hereby, attack vectors such as information withholding [40] and information eclipsing [1], as well as exploiting the rational behavior of participants [25], have received

V. Atluri et al. (Eds.): ESORICS 2022, LNCS 13554, pp. 46–66, 2022.
https://doi.org/10.1007/978-3-031-17140-6_3

particular attention. The second category of double-spending attacks leverages inadequately chosen security parameters by merchants, i.e., they provide goods or services while the probability of the payment transaction being reverted is non-negligible [27,46]. In this regard, the probabilistic consensus guarantees of Nakamoto consensus [42] may be misunderstood in practice, which contributes to insecure behavior by its users [34,46]. Regardless of the attack category, it is predominantly assumed that the adversary proactively performs double-spending through *equivocation* [22], i.e., by creating mutually exclusive transactions.

We hereby challenge this status quo and discuss an alternative attack, which we refer to as *opportunistic algorithmic double-spending*, whereby the intent to double-spend is intentionally encoded as part of the transaction semantics. Algorithmic double-spending *does not require equivocating* transactions and is facilitated through distributed ledgers that exhibit two properties, namely i) the ability to define transaction semantics that dynamically depend on the ledger state or execution context, which we refer to as *semantic malleability*, and ii) *probabilistic consensus decisions*, i.e., protocols without finality, or where security failures have compromised the *safety* of consensus decisions.

If these two conditions are fulfilled, *OpAl* can be used as a *free riding gadget* to profit from any sufficiently deep blockchain fork. *OpAl* attacks do not stand in contradiction to the security guarantees and desirable properties [18,42] offered by Nakamoto-style distributed ledgers. The existence of state instability, i.e. forks, is abstracted away in *idealized ledgers* by waiting sufficiently long for the relevant actions, e.g. transactions, to be included in the common prefix with high probability [4]. However, determining the correct choice of security parameters for real-world system settings is difficult [23,46] and unforeseen technical failures, or attacks, that undermine a ledger's security assumptions through deep forks, can happen in practice [31]. Especially during such extraordinary events the threat of *OpAl* attacks can prove particularly severe. Even under the assumption that the ledger's security guarantees hold, algorithmic double-spending can be of concern in cases where users exhibit an insecure interaction model, referred to as *hasty players* [4], whereby actions are taken based on unstable state. We crucially note that such patterns are commonly encountered in real-world ledgers such as Ethereum, e.g., in the context of *decentralized finance* (DeFi), where hastiness can be financially advantageous [9,56]. Our empirical analysis of Ethereum transactions in Sect. 6 also reveals that *OpAl*-like semantics are being used by entities which, according to block explorers, may be associated with MEV (miner extractable value) bots.

1.1 Related Work

Beyond the related work on double-spending that we mention in the introduction, it is important to note that prior art has identified a range of security issues in distributed ledgers that tie-into the discussion of *OpAl*, e.g., *timestamp- and transaction-order dependence* [32], *concurrency* and *event ordering* (EO) vulnerabilities [30,45], *blockchain anomalies* [39], *stake bleeding* [19], *time-bandit* [9] attacks, and *order-fairness* [28,55]. We outline several of these works in detail

within the body of this paper. To the best of our knowledge, we are the first to present the concept of algorithmic double-spending and demonstrate its practicability. Conceptually, Botta et al. [4] relates most to the topics discussed within this work. They effectively highlight the possible effects of blockchain forks, as well as the practical implications of probabilistic finality with *hasty players*, in the context of MPC protocols executed atop distributed ledgers. However the concept of algorithmic double-spending is not considered.

1.2 Paper Structure

An introduction, an executive summary that outlines the concept of *OpAl* and highlights the contributions of this paper, as well as background literature is presented in Sect. 1. Section 2 provides a definition of what is meant by *OpAl*. To gain a better understanding of the principles behind *OpAl*, we first define prerequisites and properties of semantic malleability in Sect. 3, and use them to investigate three different ledger designs (Sects. 4 and 5). A proof-of-concept *OpAl* attack in the context of Ethereum is also presented in Sect. 5. In Sect. 6 we empirically analyze transaction traces from Ethereum to identify and characterize transactions where ledger context is accessed. Finally, we consider possible mitigation strategies against algorithmic double-spending (Sect. 7) and highlight future research directions in Sect. 8.

2 What is Algorithmic Double-Spending?

In this section we revisit and define double-spending and propose that there exists the overlooked class of algorithmic double-spending, which does not necessitate conflicting actions, i.e., equivocation. We then discuss the implications, such as the possibility of *unintentional* double-spending, and raise the question whether double-spending requires economic damage. We observe that while research on double-spending provides concrete descriptions and formal analyses of particular instantiations of double-spending attacks, e.g., [22,27], a general definition of double-spending appears to be outstanding. A clearer definition may not only aid with classification efforts, but could also help identify new or overlooked attack forms. Motivated by this novel class of algorithmic double-spending attacks we present within this work, we hereby set out to propose such a more general definition:

Definition 1 (Double-Spending Attack). *In a double-spending attack, an adversary attempts to deceive a victim into performing an economic transaction directed at the adversary on the basis of a presumed valid system state, which is later revealed to be stale or invalid. Hereby, the adversary's goal is to be able to reuse any of the resources that form the basis of the economic transaction for other purposes. We distinguish between the following double-spending attacks:*

- ***Equivocation-Based**, whereby the adversary issues multiple conflicting actions in the system, one of which is aimed at fooling the victim, and where at most one of the issued actions can eventually be performed in the system.*

– ***Algorithmic**, whereby the adversary performs a single action that can have different semantic meanings, depending on the system state in which they are interpreted, and where the interpretation of this action in some stale or invalid system states can be used to deceive the victim.*

At the core of this work lies the insight, that double-spending may be facilitated through other means than the classical notion of equivocation-based conflicting actions by an adversary. *OpAl* builds on a simple property that can be observed in various real-world distributed ledger designs with expressive transaction semantics: *Given a transaction t, it may have different semantic outcomes, depending on the ledger state and environment upon which t is executed.*

We refer to this property as *semantic malleability* due to the fact that external factors, such as the consensus protocol and its ordering guarantees [28,55], as well as other actors in the system who may be *rushing*, e.g., in the context of frontrunning [9,14,56], are able to transition the state in a way that is able to *malleate* the intended semantics of transactions. From this observation, we can rather intuitively derive a basic strategy for an algorithmic double-spending attack: An adversary can encode both, the regular payment to the merchant, as well as an alternative malicious action, e.g., payment to herself, as different execution paths within a single transaction. The control flow of the transaction is designed to conditionally branch, depending on the ledger state σ at the time the transaction is processed by a miner. If the same transaction is included in a different state σ', i.e., a fork, the "hidden" algorithmic double-spend is triggered without active participation from the attacker. Figure 1 illustrates this difference to equivocation-based double-spending.

The concept of algorithmic double-spending raises interesting challenges, two of which we outline in more detail. First, up until now *unintentional* double-spending, for example as a result of technical failures, did not appear of particular concern. Prior art identifies potential vulnerabilities that arise from order dependence in smart contracts [30,32,45] and violations of transaction causality in forks that can have unintended side-effects, which relate to the *Paxos anomaly* [39]. We expand upon these insights by highlighting that semantic malleability can lead to unintentional algorithmic double-spending as a result of unanticipated transaction reordering that causes state changes within a blockchain fork. Hereby, it is difficult to distinguish between an intentional attack or unfortunate circumstances.

Second, in stateful smart contract systems double-spending may not only be performed solely at the economic level through coin-reuse. For example, Botta et al. [4] highlights the need for mitigation strategies against an adversary leveraging forks in MPC protocols with hasty players. In this regard, double-spending attacks can be aimed at biasing the outcome of a MPC, which may not be quantifiable in terms of economic gain. Similarly, increasing the miner fee of a transaction may require a user to equivocate, raising the question if such behavior should be subsumed under the notion of double-spending. This presents the interesting problem how any divergent system behavior within forks, be it through equivocation- or algorithmic double-spending, should be addressed if it is not

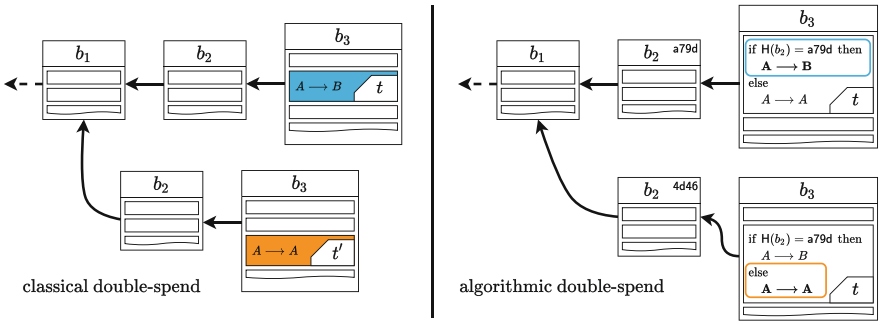

Fig. 1. Conceptual difference between equivocation- and algorithmic double-spending. Notice that in the former case $t \neq t'$ while in the latter case $t = t$.

immediately apparent that they were intended for unjust economic gain. Notice that in our Definition 1 for double-spending, we assume some economic transaction from the victim to the adversary.

3 System Model and Assumptions

Within this section, we identify prerequisites and underlying properties that enable algorithmic double-spending. Our analysis is based on an intentionally simple system model to accommodate different ledger designs. We define the concept of *semantic malleability* that we introduced in Sect. 2 and argue that ledgers with semantically malleable transactions are vulnerable to algorithmic double-spending, and thus *OpAl* attacks. In our analysis, we show that any distributed ledger that is robust to semantic malleability must satisfy two necessary properties, namely, *eventual replay validity* and *replay equivalence*.

Following Luu et al. [32], we conceptually view a blockchain as a transaction-based RSM, where its state is updated after every transaction. We denote $\mathcal{S}$ the set of all possible system states and $\sigma \in \mathcal{S}$ a single system state. The initial starting state of a blockchain is defined as σ_0. A valid transition from state σ to σ', via transaction t, is denoted as $\sigma \xrightarrow{t} \sigma'$. $\textsc{past}(\sigma_n)$ is defined as the ordered list of transactions $\mathcal{T} = (t_1, t_2, \ldots, t_n)$, that, when applied to σ_0, lead to state σ_n. If there exists a non-empty sequence of transactions starting from state σ_a to state σ_b, we call σ_a a predecessor of σ_b, in short $\sigma_a \prec \sigma_b$. The predicate $\textsc{valid}(t, \sigma)$ represents the transaction validation rules of the protocol and returns $\textsc{True}$ iff the transaction t is considered valid (executable) in state σ. We assume that block producers, e.g., miners, adhere to protocol rules and *transaction liveness* is guaranteed, i.e., any valid transaction will eventually be executed.

Executing a transaction t in state σ alters (part of) the state σ and thus results in a new state σ'. The changes are captured by the function $\textsc{diff}(t, \sigma)$. For example, consider a state $\sigma = \{\text{Alice: } 6, \text{Bob: } 5, \text{Carol: } 4\}$ represented as an account-value mapping, and a transaction t, where Alice gives 2 coins to Bob.

Then $\text{DIFF}(t, \sigma) = \{\text{Alice: } -2, \text{Bob: } +2\}$ captures the balance changes of Alice and Bob while other parts of the state (Carol's balance) remain unaffected. In this example a single account-value mapping is called a *substate*. Note that it is possible that the effects of executing the same transaction t in two different states are equal, i.e., $(\sigma_a \neq \sigma_b) \wedge (\text{DIFF}(t, \sigma_a) = \text{DIFF}(t, \sigma_b))$.

We consider a transaction t to be a sequence of operations (computations) that lead to a state transition. A transaction is *semantically malleable*, if the available operations, which are used to define the semantics of the transaction, allow the control flow of the execution to branch conditionally based on the particular input state σ. The following two properties we define are necessary, but not sufficient, for a ledger to be robust against semantic malleability. We refer to these properties as *replay equivalence* and *eventual replay validity*. Replaying the same ordered set of transactions on some initial state σ_0 should always yield the same state transitions and final state, and the validity of transactions should not be affected by the environment.

Definition 2 (replay equivalence). *Assuming that no transaction equivocation happens: A transaction t satisfies replay equivalence, if executing t in all candidate states where t is executable (valid) leads to the same changes in the respective (sub)states:*

$$\begin{aligned} &\forall \sigma_a, \sigma_b \in \mathcal{S}, \\ &(\text{VALID}(t, \sigma_a) \wedge \text{VALID}(t, \sigma_b)) \implies (\text{DIFF}(t, \sigma_a) = \text{DIFF}(t, \sigma_b)) \,. \end{aligned} \tag{1}$$

Definition 3 (eventual replay validity). *Assuming that no transaction equivocation happens: If a transaction t is found executable (valid) in some state σ_a, then it either remains executable (valid) or has already been executed in predecessor states of σ_a:*

$$\begin{aligned} &\forall \sigma_a, \sigma_b \in \mathcal{S}, \\ &(\text{VALID}(t, \sigma_a) \wedge \sigma_a \prec \sigma_b) \implies (t \in \text{PAST}(\sigma_b) \vee \text{VALID}(t, \sigma_b)) \,. \end{aligned} \tag{2}$$

Definition 4 (semantic malleability). *A transaction t is semantically malleable if it violates replay equivalence and/or eventual replay validity.*

4 Semantic Malleability of Bitcoin and Cardano

For the following investigation, we set aside the orthogonal topic of *how* to create blockchain forks of sufficient depth to facilitate double-spending attacks. Instead, we are interested in identifying if, in principle, the designs are vulnerable to semantic malleability by evaluating whether the aforementioned necessary properties are violated. We first consider Bitcoin and Cardano within this Section, and then cover Ethereum separately in Sect. 5. Each ledger represents an instantiation of a Nakamoto-style blockchain with distinct design differences. Bitcoin [38] is UTXO based and facilitates a highly limited, non-Turing complete scripting language for transaction semantics [2]. Cardano [8] adopts the

EUTXO model [5], which leverages advantages of a stateless UTXO design with the expressiveness of Turing-complete smart contracts that can carry state.

Bitcoin: In Bitcoin, transactions are based on the so-called *unspent transaction outputs* (UTXO) model [11] and contain simple (deterministic) Boolean functions, called *Scripts*, that determine the transaction semantics [2]. Bitcoin's UTXO model is stateless and non-Turing complete. A key aspect of the UTXO model is that transactions are deterministic and bound to a single execution by committing to the exact input (sub)states, i.e., UTXOs, that a transaction consumes, and a precise set of output UTXOs, that the transaction produces.

Furthermore, within Bitcoin transactions the access to external ledger state is not made explicitly by including it as an input in the transaction, but implicitly through Scripts or when defining the validity of the transaction in terms of the block height or current time at the protocol level. There currently exist only a limited number of primitives that can be used to constrain the validity of a transaction to some external context. Specifically, it is possible to define some relative or absolute time, in relation to that of the ledger context, from which point onward a transaction may become *valid* [47]. However, *it is not possible to permanently invalidate a previously valid transaction that depends on ledger context*, i.e., in a live blockchain, there is a future point in time where this dependency is satisfied. Therefore, in principle, the Bitcoin UTXO model could satisfy eventual replay validity. However, we show that in case of deep forks, eventual replay validity can be violated by coinbase transactions, making Bitcoin-like UTXO cryptocurrencies theoretically vulnerable to semantic malleability.

Theorem 1 (Semantic malleability of Bitcoin-like UTXO cryptocurrencies with coinbase transaction). *A Bitcoin-like UTXO based cryptocurrency is affected by semantic malleability if it programmatically allows the issuance of special per-block transactions as payout, i.e., coinbase transactions, transferring collected fees and/or rewards for block creation.*

Proof. We show that A Bitcoin-like UTXO cryptocurrency is affected by semantic malleability by constructing a counterexample violating the *eventual replay validity* property: Let σ_a be some blockchain state and $t_c \neq t'_c$ two different coinbase transactions (e.g., rewarding different miners) that are valid in this state if included by a newly mined block, i.e., $\textsc{Valid}(t_c, \sigma_a) \wedge \textsc{Valid}(t'_c, \sigma_a)$. Let there be a blockchain with a new block containing t_c st. $\sigma_a \xrightarrow{t_c} \sigma_b$ and thus $\sigma_a \prec \sigma_b$. In Bitcoin-like UTXO cryptocurrencies, the coinbase transaction can only be issued at the beginning of each block and is tied to the respective block height[1]. Therefore, the other coinbase transaction t'_c cannot be included anymore in state σ_b. The reason for this is that executing the block containing t_c (and potentially other transactions) necessarily leads to a state σ_b with increased block height. Therefore, there exists a σ_b st. $t'_c \notin \textsc{past}(\sigma_b) \wedge \neg\textsc{Valid}(t'_c, \sigma_b)$ which violates *eventual replay validity.* □

[1] cf. https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki.

In practice, the potential consequences of the semantic malleability of coinbase transactions are mitigated by the maturation period of 100 blocks, after which transactions can be included that spend coinbase UTXOs. As an attack example, consider a transaction with an output from a recent coinbase transaction that is spendable (i.e., has matured for 100 Blocks) as one of its input UTXOs. If a sufficiently deep blockchain fork, of say 144 blocks, occurs and this coinbase transaction does not exist in the chain anymore, the depending transaction using its UTXO as input can not be replayed within a fork and becomes invalid. Therefore, coinbase transactions could facilitate algorithmic double-spending.

The design of forkable Nakamoto-style cryptocurrencies, which provide payouts in terms of fees/block rewards to incentivize participation, necessarily require payments depending on the state of the blockchain, i.e., context, which inherently violates eventual replay validity. Thus, in Sect. 7.1, we raise the question whether characterizing Nakamoto-style ledgers as replicated state machines (RSM) is accurate in light of algorithmic double-spending.

Cardano: Cardano [8] is based on a line of research on provably secure proof-of-stake Nakamoto-style blockchains [3,10,29], which we subsume under the term *Ouroboros*. Ouroboros, as it is currently realized in Cardano, offers probabilistic finality guarantees and the existence of temporary blockchain forks is possible. Cardano adopts the *Extended UTXO* (EUTXO) model [5,7], that was conceived to leverage desirable properties of Bitcoin's UTXO design for more expressive transaction semantics [7]. Conceptually, to support stateful Turing-complete smart contracts in EUTXO, the UTXO model is extended in the following (from Chakravarty et al. [7]): i) outputs can contain arbitrary contract-specific data; ii) Scripts, which are referred to as *validators* in the EUTXO model, receive the entire transaction information, including its outputs, as *context* next to the contract specific data, and can impose arbitrary validity constraints on the transaction; iii) a *validity interval* is added for transactions, which is specified as an interval of "ticks"[2], whereby any Scripts which run during validation can assume that the current time is within that interval, but do not know the precise value;

A key property the EUTXO model inherits from the UTXO model is that the execution of a transaction during validation is entirely deterministic and solely determined by its inputs. Equivocation is hence required to achieve a different semantic result. In terms of our necessary properties to achieve robustness against semantic malleability, *replay equivalence* follows analogous to Bitcoin.

However, as Brünjes and Gabbay [5] crucially point out, the EUTXO model allows restricting the validity of transactions to time intervals, which renders the result of transaction processing dependent on the ledger context. Unlike Bitcoin, in Cardano transactions can be permanently invalidated based on ledger context. Hence, *eventual replay validity* is not satisfied and semantic malleability possible.

Corollary 1 (Semantic malleability of Cardano-like EUTXO cryptocurrencies that support limited validity transactions). *A EUTXO*

[2] [7] assume that in practice a tick will correspond to a block number or block height.

based cryptocurrency is affected by semantic malleability, if it programmatically allows the issuance of limited validity transactions which are valid at some point in the chain, but become invalid after a certain block height or time interval.

Proof. We show that Cardano-like EUTXO cryptocurrencies that support limited validity transactions are semantically malleable, by pointing out that the desired properties of such transactions directly negate and thus violate *eventual replay validity*. Let t_v be a limited validity transaction and σ_a be some blockchain state where $\textsc{Valid}(t_v, \sigma_a)$, which is true when the specified criteria (block height or time) is satisfied. By definition of a limited validity transaction, there must exist a state $\sigma_a \prec \sigma_b$ st. $\neg\textsc{Valid}(t_v, \sigma_b)$. Due to forks, or congestion, it might be the case, that t_v is not included until σ_b is reached, thus $t_v \notin \textsc{past}(\sigma_b)$. Therefore, t_v is invalidated after this point and cannot be included in any other subsequent block. Hence, by the construction of limited validity transactions,

$$\exists \sigma_a, \sigma_b \left(\textsc{Valid}(t_v, \sigma_a) \land \sigma_a \prec \sigma_b \land t_v \notin \textsc{past}(\sigma_b) \land \neg\textsc{Valid}(t_v, \sigma_b)\right),$$

which is exactly the negation of our definition of *eventual replay validity*. □

As an example, consider a payment transaction to a merchant where the validity is constrained to a specific block height. Thus, an *OpAl* attack is triggered if the transaction does not make it into a block in time during a fork.

5 Semantic Malleability in Ethereum

Ethereum [51] adopts an account-based model and offers expressive transaction semantics that can draw upon stateful Turing-complete smart contract functionality. Due to the various ways in which replay equivalence and eventual replay validity can be violated in Ethereum, we omit a formal analysis and directly discuss a proof-of-concept (PoC) *OpAl* attack and its practical implications.

Our attack design is inspired, on the one hand, by *hardfork oracles*, which McCorry et al. [35] discusses in the context of atomic-trade protocols during hardforks, and, on the other hand, by the notion of *context sensitive transactions* Gaži et al. [19] describes as a replay protection mechanism in *stakebleeding attacks*. An informal statement that encapsulates the intended transaction semantics for our PoC *OpAl* attack is the following:

> "IF *this transaction is included in a blockchain that contains a block with hash* $0xa79d$ THEN *pay the merchant,* ELSE *don't pay the merchant."*

Essentially, our attack is based on the insight that a transaction can act as its own *fork oracle* for conditionally branching its execution. In the following, we first outline the construction of such a fork oracle in more detail and then present a PoC attack that allows transactions with the above semantics to be created.

5.1 How to Construct an *OpAl* Fork Oracle in Ethereum

The concept of employing a fork oracle to distinguish between branches of (hard)forks was proposed in cryptocurrency communities [15,33], as well as research [24,35,36]. Hereby, a frequent goal is achieving *replay protection*. McCorry et al. [35] outlines how fork oracles can be leveraged to realize atomic trades across hardforks. Constructing a smart contract based fork oracle if the underlying forks do not offer replay protection can be challenging [35]. McCorry et al. [36] demonstrate through *history revision bribery* how (equivocation-based) double-spending can be leveraged to realize a fork oracle for a smart contract based bribing scheme for incentivizing forks. Hereby, the fork oracle is not used to facilitate (algorithmic) double-spending. Rather, the mutually exclusive outcomes of the double-spend in different forks are relied upon to actually implement the oracle. Surprisingly, to the best of our knowledge, the idea of using fork oracles to *algorithmically* trigger double-spending was not yet considered.

Block-Hash Based Fork Oracle. The fork oracle we propose is inspired by a simple and elegant technique to achieve replay protection considered in the proof-of-stake (PoS) setting [19]. Hereby, the hash of a recent block is included in a transaction, and it is only considered valid for blockchains that contain this block in their prefix. [19] refer to this mechanism as *context sensitive transactions*. Essentially, context sensitive transactions already implicitly realize the attack semantics described above.[3] In case a fork of sufficient depth occurs, this replay protection mechanism ensures that transactions become invalid at the protocol level, and the double-spending "attack" is realized algorithmically through the underlying protocol rules. Ethereum does not natively support context sensitive transactions, however, this functionality can be emulated with smart contract code using EVM primitives that expose ledger context, such as the BLOCKHASH opcode [51]. It is hence possible to programmatically act upon the existence of a particular block, or other ledger context, as part of an Ethereum transaction.

Fork Oracle Discussion. A downside of hash-based fork oracles is the reliance on a commitment to *previous* ledger state, thereby requiring a fork of at least depth-2 to trigger the attack. However, it is also possible to construct oracles for forks of depth-1. The key difference between a depth-1 fork oracle and a hash-based fork oracle is that the latter is based on ledger state which is *known*, whereas the former is based on some *prediction* of the future state at the time the transaction is processed. Hence, depth-1 fork oracles generally offer weaker probabilistic guarantees for identifying forks. For example, consider the EVM COINBASE opcode that returns the *current* block's beneficiary address [51]. An adversary could use the beneficiary address of a large mining pool in a depth-1 *OpAl* attack. Hereby the transaction semantics depend on whether the transaction is included in a block from the targeted mining pool or some other miner.

[3] Thereby introducing the possibility of *unintentional OpAl* attacks (see Sect. 2).

Generally speaking, in Nakamoto-style proof-of-work ledgers the next block producer is not known in advance. However, we note that in some PoS protocols this can be different [44], allowing for more reliable depth-1 fork oracles.

Another limitation of the hash-based fork oracle specific to the EVM is the restriction that the BLOCKHASH opcode only returns hashes within a 256 block lookback window, and 0 otherwise [51]. Hence, if a transaction is processed in a block that exceeds 257 blocks after the height of the blockhash commitment, the oracle will falsely report a fork and trigger the attack branch. We argue that in the case of our intended *OpAl* semantics this limitation is unproblematic, as the transaction would simply transfer the funds back to the attacker.

```
pragma solidity 0.8.4;
// This contract acts as an OpAl forwarding proxy for transactions.
contract Opal {
  address public owner;
  modifier onlyOwner() { require(isOwner(msg.sender)); _; }
  constructor() { owner = msg.sender; }
  fallback() external payable {}
  receive() external payable {}
  function isOwner(address addr) public view returns(bool)
  { return addr == owner; }
  function cashOut(address payable _to) public onlyOwner
  { _to.transfer(address(this).balance); }

  // forwarding function implementing opportunistic double-spending (OpAl)
  function forward(address payable destination, bytes32 commitblockHash,
                   uint commitblockNumber, bytes memory data)
                   onlyOwner public payable returns(bool success) {
    if (blockhash(commitblockNumber) == commitblockHash)
      assembly { success := call(gas(), destination, callvalue(),
                                 add(data, 0x20), mload(data), 0, 0)
                                       }
  }
}
```

Listing 1.1. Solidity *OpAl* contract that implements a basic fork oracle by only forwarding transactions if the provided commitment to a block hash can be resolved.

5.2 Proof of Concept *OpAl* Attack Contract

Di Angelo and Salzer present a comprehensive empirical analysis of wallet contracts on Ethereum [12]. Of the identified properties, in particular, designs that support *flexible transactions*, i.e., forwarding of arbitrary calls, appear suitable for augmentation to support the creation of *OpAl* transactions. Their empirical data shows that at least *tens of thousands* of contracts supporting flexible transactions are currently deployed in Ethereum, suggesting practical use-cases for such contract patterns, even without an *OpAl* augmentation. Our attack requires minimal modifications, and the interaction pattern is almost identical.

In the following, we present a minimal *fully viable PoC OpAl attack smart contract* written in Solidty [50], that relies on the aforementioned hash-based fork oracle. Our contract code (Listing 1.1) is loosely based on the Executor contract from the Gnosis-Safe Wallet [37], which allows the forwarding of arbitrary func-

tion calls. Instead of forwarding a call directly, the contract first evaluates if the block hash at a particular height of the current ledger matches the commitment hash that is provided as an additional parameter in the transaction data. This is realized through the *blockhash()* function [51]. If the blockhash matches the commitment, the function call is forwarded. Else, no action is performed, i.e., the action is reversed whenever the transaction is replayed in a fork.

Outline of the Attack. An adversary wishing to engage in *OpAl* first needs to deploy the attack contract. Once the contract is successfully deployed, whenever they wish to perform a transaction with *OpAl* functionality, instead of calling a function $f()$ in the target contract or sending funds directly, they simply forward this call to the *forward()* function (Line 15 in Listing 1.1) of the deployed attack contract, together with the appropriate parameters. Specifically, the adversary generates transaction t that calls *forward* in the attack contract with the following parameters: i) the target address; ii) the block hash and height h of the current chain tip; iii) the encoded function name to be called at the target $f()$ together with its parameters; iv) any Ether that shall be sent; and broadcasts t to the network. Ideally, the transaction fee is high enough for t to be immediately included in the next block $h+1$. Otherwise, the required fork depth increases in the number of blocks the chain grows between the creation and inclusion of t.

To the recipient of t, the interaction pattern will appear as if the user employed a regular wallet contract. Unless they perform an analysis of the execution trace, the malicious behavior only becomes apparent once the attack conditions are triggered, i.e., during a fork. In case the adversary is lucky and a fork at, or before, height h occurs, and their transaction is replayed within this fork, the alternative attack branch of the contract is executed automatically.

5.3 Cost Overhead of PoC Attack in Ethereum

We quantify the additional costs incurred when augmenting a transaction with *OpAl* capabilities by deploying our attack contract in a private Ethereum testnet and measuring the gas utilization for basic interactions, such as ERC-20 token [49] transfers. Our PoC *OpAl* attack adds a constant overhead of gas that depends on the number of parameters supplied to the target function $f()$. The deployment transaction for the contract in Listing 1.1 required 393 175 gas. As it is not essential for the contract to be deployed in a recent block, and can be done well in advance of any attacks, we assume a gas price of 50 GWei, which translates to deployment costs of ≈0.02 Ether or, at an exchange rate of 2 000 USD, approximately 40 USD. Note that this contract needs to be deployed once, after which the only overhead derives from using the forwarding function. For ERC-20 token interactions (*approve, transfer, transferFrom*), using *OpAl* adds ≈3 000 gas, which equates to ≈8% overhead. At the time of writing, assuming a gas price of 100 GWei for timely inclusion[4] of the transaction, this overhead

[4] For simplicity we consider legacy transactions and omit pricing based on EIP-1559.

translates to ≈0.6 USD higher fees if a transaction is augmented to support *OpAl* attacks, rendering our attack a viable default strategy for most cases.

6 Empirical Analysis of Ethereum Transaction Traces

We empirically analyze the execution traces of 922 562 transactions from 5 000 Ethereum blocks in order to identify and characterize transactions where ledger context is accessed. Hereby, block selection for the analysis was performed in batches of 100 consecutive blocks every 1000 blocks, starting from block height 14 010 000 up to block 14 059 099 to obtain a sample spread over a wider time window. The selection of blocks for our analysis was necessitated due to the steep storage and processing requirements for analyzing full execution traces. For every considered block, we parse the execution trace of all included transactions and record whether the trace contains EVM opcodes that are characteristic for accessing the ledger context. The specific opcodes[5] that we considered are highlighted in Table 1. Our analysis reveals that 231 271 transactions, or ≈ 25%, include at least one of these opcodes, whereby roughly every 5th transaction uses `TIMESTAMP`, while the other opcodes are encountered considerably less often.

Table 1. EVM Opcode occurrence within the analyzed block range.

Opcode (OP)	`TIMESTAMP`	`SELFBALANCE`	`NUMBER`	`BALANCE`	`CHAINID`	`BASEFEE`	`BLOCKHASH`	`COINBASE`	`DIFFICULTY`	`GASLIMIT`
TX containing OP	199731	63594	36859	4324	8253	777	3425	3882	1251	906
% of TX with OP	21.65%	6.893%	3.995%	0.469%	0.895%	0.084%	0.371%	0.421%	0.136%	0.098%
Blocks cont. OP	4886	4767	4529	2265	3071	641	1830	1897	812	545
% of Blocks with OP	97.72%	95.34%	90.58%	45.3%	61.42%	12.82%	36.6%	37.94%	16.24%	10.9%

Of particular interest are transactions that include *both* `BLOCKHASH` and `NUMBER` opcodes in their traces, as this combination is also present in our PoC *OpAl* attack. We are able to identify 3 338 transactions with such an *OpAl*-like opcode signature within 1 823 (≈36%) of the analyzed blocks. Table 2 shows the top 10 contract addresses that these transactions were directed at, as well as a generalized categorization of their purpose based on publicly available information. Analyzing the decompiled[6] bytecode of the contract with the second most *OpAl*-like transaction interactions, we indeed discover an *OpAl*-like pattern. Listing 1.2 highlights the relevant code section, which, in plaintext, evaluates whether the first 4 Bytes of the previous block hash match those stored as part of the transaction data and reverts the execution otherwise. We further confirm this behavior by observing transactions to the aforementioned contract that were reverted due to an incorrect commitment[7]. While this pattern is likely intended to render the transaction *context sensitive* to prevent execution in an undesirable state, it could nevertheless be used for *OpAl* attacks simply by subsequently using the transferred/traded funds for payments to a victim.

[5] Cf. the Ethereum Yellow paper [51] for details on EVM opcodes and their behavior.
[6] Cf. https://ethervm.io/decompile/0x000000000035B5e5ad9019092C665357240f594e.
[7] Cf. txn: 0x2368617cf02cf083eed2d8691004c1ad0176976b6fa83873bc6b0fd7de4cc7fc.

Table 2. Contracts with the highest number of transaction interactions with EVM opcodes that are also characteristic of *OpAl*. (?) denotes uncertain categorizations.

Contract address	TX int.	Purpose	Name	Source	Opcode purpose
0xc5F85281d4402850ff436b959a925a0e811D78d3	557	Game/Token	CnMGame	yes	randomness?
0x0000000000035B5e5ad9019092C665357240f594e	411	MEV Bot?	?	no	context sensitivity?
0xEef86c2E49E11345F1a693675dF9a38f7d880C8F	313	MEV Bot?	?	no	context sensitivity?
0x5E4e65926BA27467555EB562121fac00D24E9dD2	264	Layer 2 rollup	optimism.io	yes	caching/processing
0x56a76bcC92361f6DF8D75476feD8843EdC70e1C9	227	Layer 2 rollup	metis.io	yes	caching/processing
0xB6eD7644C69416d67B522e20bC294A9a9B405B31	222	Token	0xbitcoin.org	yes	context sensitivity
0xd6e382aa7A09fc4A09C2fb99Cfce6A429985E65d	221	Game/Token	Doomsday NFT (BUNKER)	yes	randomness
0x75E9Abc7E69fc46177d2F3538C0B92d89054eC91	130	Token/NFT	EnterDAO Sharded Minds	yes	randomness
0x563bDabAa8846ec445b25Bfbed88d160890a02Ed	115	MEV Bot?	?	no	context sensitivity?
0xa10FcA31A2Cb432C9Ac976779DC947CfDb003EF0	111	MEV Bot?	?	no	context sensitivity?

```
function func_060C() {
  if (msg.data[0x04:0x24] >> 0xe0 ==
      block.blockHash(block.number + ~0x00) >> 0xe0) { return; }
  // ... code omitted for brevity
  revert(memory[0x60:0xc4]); }
```

Listing 1.2. Code snippet from decompiled contract (tagged as MEV bot) showing *OpAl*-like pattern. Notice that `~0x00` corresponds to -1 in Two's complement.

7 Mitigation Strategies Against *OpAl*

Having outlined the principles behind algorithmic double-spending, we now discuss possible prevention or mitigation strategies. Hereby, we broadly distinguish between two categories: i) Approaches that address instability in consensus, i.e., a *lack of finality*. ii) Approaches that seek to limit the effects of *semantic malleability*. Finally, we discuss if the characterization of blockchains as replicated state machines is accurate in light of semantic malleability.

Mitigating OpAl Through Stronger Consensus Guarantees: Essentially, the majority of distributed ledgers rely on *consensus* to agree upon the order of transactions among participants in order to prevent double-spending [21]. Thus, one possible defensive approach against *OpAl* attacks is to prevent players from concurrently interacting with malleable ledger state until it is sufficiently stable. In this regard, it appears advantageous to achieve fast and guaranteed *consensus finality*, which remains an active research topic for decentralized ledger designs [41]. Our Definition (Definition 1) of double-spending highlights the requirement of some stale or invalid system state in order to fool a victim. The existence of hasty players who are willing to act upon such state renders double-spending attacks feasible in practice, even if the consensus protocol in principle could provide stronger guarantees. In this regard, effective mitigation strategies to combat double-spending may also entail the stricter enforcement of safe interaction patterns in client software and cryptocurrency wallets, and a better understanding of the behavior and mental models of cryptocurrency users [34].

However, if the security assumptions of the underlying system are compromised, in particular, Nakamoto-style distributed ledgers can suffer from deep forks where previously assumed stable ledger state is reverted. Aside from the potential of targeted attacks against the protocol [1,48], technical failures[8] can also lead to such a violation of the security assumptions [31,35]. Notice that in this regard there is a crucial difference between *OpAl* and equivocation-based double-spending. In the latter, an adversary has to actively monitor the network for forks and disseminate conflicting double-spending transactions that are at risk of being easily detected and prevented at the peer-to-peer layer [20,27]. *OpAl* attacks and algorithmic double-spending, on the other hand, may prove particularly severe. Any transaction that was included in a blockchain that is replayed on a fork faces the risk of triggering a hidden *OpAl* attack. If a fork in excess of k blocks occurs, triggered *OpAl* attacks have a high probability of success. A possible mitigation strategy to limit the effects of *OpAl* in deep forks is the utilization of checkpointing [26]. Another line of research seeks to strengthen the guarantees of Nakamoto consensus by achieving consensus finality [13,41]. It may also be preferable to sacrifice *liveness* by halting execution rather than risking systemic risk through *OpAl* attacks.

Mitigating Semantic Malleability: As we have shown in Sects. 2 and 3, semantic malleability lies at the core of enabling algorithmic double-spending. Semantically malleable transactions allow for different state transitions, depending on the input state and execution environment at the time of processing – a property that is generally observed within smart contract platforms. In this regard, we believe that the expressive transaction semantics associated with smart contract functionality poses a fundamental challenge when trying to combat algorithmic double-spending. Drawing upon the concept of *guard functions* from Luu et al. [32] and *context sensitive transactions* Gaži et al. [19] and Botta et al. [4] rely on, transaction validity should more explicitly be constrained to input states that only lead to desirable outcomes for the sender. While such patterns do not prevent the possibility of algorithmic double-spending, they can avert that a user's transaction executes in a state that leads to an undesirable outcome. In light of recent research in regard to *order-fairness* in consensus [28,55], the aforementioned pattern could also help to mitigate the potential negative impact of malicious orderings. Similar to the concept of the *Let's Go Shopping Defense* [23], a highly questionable mitigation strategy might be to oneself proactively engage in *OpAl* (counter-) attacks in order to reduce counterparty risk and try to hedge against the potentially detrimental effects of any deep blockchain fork, should it ever occur.

Another mitigation strategy by which to address semantic malleability and algorithmic double-spending is through the analysis and classification of transaction semantics, in order to try and identify potential threats and malicious behavior. Hereby, the challenges lie on the one side, in finding efficient techniques for static and dynamic code analysis that can be applied, in real-time, to identify

[8] We note that scheduled protocol updates carry a risk of unintentional forks, and an adversary may try to leverage this by performing *OpAl* transactions at that time.

potentially malicious transactions before they are processed, and on the other side, in how to define what is considered malicious behavior and also enforce any transaction rejection policies within decentralized systems [16,17,52,53].

For platforms that do not support expressive transaction semantics, it may appear that the solution to this problem is to enforce only a single valid state transition for transactions, such as the EUTXO model [7] employed by Cardano. However, in this case the possibility of algorithmic double-spending still arises if the *validity* of a transaction can be tied to particular ledger states, which is generally the case. In the UTXO model of Bitcoin [2], transaction expressiveness and access to ledger state are sufficiently constrained to prevent practicable *OpAl* attacks, apart from the possibility of using recent coinbase transactions to limit replay validity in case of deep-forks. However, since the mechanism design of most cryptocurrencies relies on the issuance of rewards to incentivize participation [6], it is unclear if the underlying issue could be completely avoided in practice.

7.1 Can Blockchains Be Characterized as State Machines?

In his seminal work on the state machine approach, F. B. Schneider provides the following semantic characterization of a replicated state machine (RSM): *"Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in a system."* [43] Interestingly, while blockchains are often considered to realize RSMs, e.g., in the model we adopt from Luu et al., we observe (Sect. 4) that *in practice*, ledger designs appear to actually deviate from this characterization.

First, consider the herein discussed property of semantic malleability in transactions. Semantic malleability in itself does not violate the above characterization, as a mere reordering of transactions, i.e., requests, may lead to semantic malleability without requiring any access to time or activity within the system. However, in practice, ledger designs often allow transaction semantics to depend on external ledger context that is not solely defined by such requests, i.e., *time* or other *external data* (See Sect. 5). In essence, being able to define functions that can act upon such context within transaction semantics, such as previous block hashes, the block height, coinbase transactions, or block time, can cause a violation of replay equivalence or eventual replay validity, both of which can be directly derived as required properties of a RSM from the above characterization.

Second, blockchain designs generally offer *rewards* as an incentive mechanism for block producers to participate in the consensus protocol. Under the assumption that a block merely represents an ordered set of transactions, i.e., requests, and transactions can not access any external state defined within blocks, this model would appear to realize a RSM. However, if we include the fact that block rewards represent transactions or state transitions that depend on a particular external state, namely the block itself that justifies the reward, the model is no longer independent of the system state.

We note that one possibility to amend this issue is to either include the creation of blocks as requests, or model state updates entirely from the perspective of blocks and not at the transaction level. The latter approach is, for instance,

taken by formal models that analyze Nakamoto consensus [18,42]. Nevertheless, even if one considers state machine replication only from the perspective of blocks and not individual transactions, there can still exist external dependencies on the environment, in particular on time. Consider that receiving late or early blocks may render them (temporarily) invalid by the protocol rules, leading to different possible interpretations of the same sequence of requests and resulting final state depending on the current system time.

8 Conclusion

We have described and analyzed a novel class of double-spending attacks, called (opportunistic) algorithmic double-spending (*OpAl*), and shown that *OpAl* can readily be realized in stateful smart contract platforms, by presenting a proof-of-concept implementation for EVM-based designs. *OpAl* itself does not increase the likelihood or severity of blockchain forks, which are a prerequisite for most double-spending attacks. Instead, *OpAl* allows regular transactions performed by *anyone* to opportunistically leverage forking events for double-spending attacks while evading common detection strategies and offering a degree of plausible deniability. A particularly worrying property of *OpAl* is the ability for already processed transactions to trigger hidden double-spending attacks whenever they are replayed in a fork. Hereby, our empirical analysis of 922 562 transaction traces in Ethereum reveals that transactions with *OpAl*-like semantics already exist in practice. While these transactions are likely intended for a different use case, the effect in case of a fork could still lead to unintentional double-spending. Attacks or technical failures that lead to deep forks may hence pose an even greater systemic risk than previously assumed. It would appear that the most promising mitigation strategy against *OpAl* is achieving fast consensus finality, combined with avoiding semantic malleability in transactions.

Acknowledgements. This material is based upon work partially supported by (1) the Christian-Doppler-Laboratory for Security and Quality Improvement in the Production System Lifecycle; The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the Nation Foundation for Research, Technology and Development and University of Vienna, Faculty of Computer Science, Security & Privacy Group is gratefully acknowledged; (2) SBA Research; the competence center SBA Research (SBA-K1) funded within the framework of COMET Competence Centers for Excellent Technologies by BMVIT, BMDW, and the federal state of Vienna, managed by the FFG; (3) the FFG Industrial PhD projects 878835 and 878736. (4) the FFG ICT of the Future project 874019 dIdentity & dApps. (5) the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078 (FeatureCloud). We would also like to thank our anonymous reviewers for their valuable feedback.

References

1. Apostolaki, M., Zohar, A., Vanbever, L.: Hijacking bitcoin: routing attacks on cryptocurrencies. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 375–392. IEEE (2017)
2. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of bitcoin transactions. In: Meiklejohn, S., Sako, K. (eds.) FC 2018. LNCS, vol. 10957, pp. 541–560. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-58387-6_29
3. Badertscher, C., Gaži, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros genesis: composable proof-of-stake blockchains with dynamic availability. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 913–930 (2018)
4. Botta, V., Friolo, D., Venturi, D., Visconti, I.: Shielded computations in smart contracts overcoming forks. In: Financial Cryptography and Data Security-25th International Conference, FC, pp. 1–5 (2021)
5. Brünjes, L., Gabbay, M.J.: UTxO- vs account-based smart contract blockchain programming paradigms. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12478, pp. 73–88. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_6
6. Carlsten, M., Kalodner, H., Weinberg, S.M., Narayanan, A.: On the instability of bitcoin without the block reward. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 154–167. ACM (2016)
7. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Peyton Jones, M., Wadler, P.: The extended UTXO model. In: Bernhard, M., et al. (eds.) FC 2020. LNCS, vol. 12063, pp. 525–539. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54455-3_37
8. Corduan, J., Vinogradova, P., Gudemann, M.: A formal specification of the cardano ledger (2019)
9. Daian, P., et al.: Flash boys 2.0: frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 910–927. IEEE (2020)
10. David, B., Gaži, P., Kiayias, A., Russell, A.: Ouroboros praos: an adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10821, pp. 66–98. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_3
11. Delgado-Segura, S., Pérez-Solà, C., Navarro-Arribas, G., Herrera-Joancomartí, J.: Analysis of the bitcoin UTXO set. In: Zohar, A., et al. (eds.) FC 2018. LNCS, vol. 10958, pp. 78–91. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-58820-8_6
12. Di Angelo, M., Salzer, G.: Wallet contracts on ethereum. In: 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pp. 1–2. IEEE (2020)
13. Dinsdale-Young, T., Magri, B., Matt, C., Nielsen, J.B., Tschudi, D.: Afgjort: a partially synchronous finality layer for blockchains. In: Galdi, C., Kolesnikov, V. (eds.) SCN 2020. LNCS, vol. 12238, pp. 24–44. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57990-6_2
14. Eskandari, S., Moosavi, S., Clark, J.: SoK: transparent dishonesty: front-running attacks on blockchain. In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) FC 2019. LNCS, vol. 11599, pp. 170–189. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43725-1_13

15. Ethereum Community: Issue#134 ethereum/eips (2016). https://github.com/ethereum/EIPs/issues/134
16. Ferreira Torres, C., Baden, M., Norvill, R., Jonker, H.: Ægis: smart shielding of smart contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2589–2591 (2019)
17. Ferreira Torres, C., Iannillo, A.K., Gervais, A., et al.: The eye of horus: spotting and analyzing attacks on ethereum smart contracts. In: International Conference on Financial Cryptography and Data Security, Grenada, 1–5 March 2021 (2021)
18. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_10
19. Gaži, P., Kiayias, A., Russell, A.: Stake-Bleeding Attacks on Proof-of-Stake Blockchains. Cryptology ePrint Archive, Report 2018/248 (2018)
20. Grundmann, M., Neudecker, T., Hartenstein, H.: Exploiting transaction accumulation and double spends for topology inference in bitcoin. In: Zohar, A., et al. (eds.) FC 2018. LNCS, vol. 10958, pp. 113–126. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-58820-8_9
21. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., Seredinschi, D.A.: The consensus number of a cryptocurrency. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, pp. 307–316 (2019)
22. Iqbal, M., Matulevičius, R.: Exploring sybil and double-spending risks in blockchain systems. IEEE Access **9**, 76153–76177 (2021)
23. Judmayer, A., Stifter, N., Schindler, P., Weippl, E.: Estimating (miner) extractable value is hard, let's go shopping! In: 3rd Workshop on Coordination of Decentralized Finance (CoDecFin) (2022, to appear)
24. Judmayer, A., et al.: Pay to win: cheap, crowdfundable, cross-chain algorithmic incentive manipulation attacks on pow cryptocurrencies (2019). https://ia.cr/2019/775
25. Judmayer, A., et al.: SoK: algorithmic incentive manipulation attacks on permissionless PoW cryptocurrencies. In: Bernhard, M., et al. (eds.) FC 2021. LNCS, vol. 12676, pp. 507–532. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-662-63958-0_38
26. Karakostas, D., Kiayias, A.: Securing proof-of-work ledgers via checkpointing. In: 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pp. 1–5. IEEE (2021)
27. Karame, G.O., Androulaki, E., Capkun, S.: Double-spending fast payments in bitcoin. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 906–917 (2012)
28. Kelkar, M., Zhang, F., Goldfeder, S., Juels, A.: Order-fairness for byzantine consensus. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 451–480. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_16
29. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: a provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 357–388. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_12
30. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the laws of order in smart contracts. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 363–373 (2019)

31. Lovejoy, J.P.T.: An empirical analysis of chain reorganizations and double-spend attacks on proof-of-work cryptocurrencies. Master's thesis, Massachusetts Institute of Technology (2020)
32. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: 23rd ACM Conference on Computer and Communications Security (ACM CCS 2016) (2016)
33. Maersk, N.: Thedaohardforkoracle (2016). https://github.com/veox/solidity-contracts/blob/TheDAOHardForkOracle-v0.1/TheDAOHardForkOracle/TheDAOHardForkOracle.sol
34. Mai, A., Pfeffer, K., Gusenbauer, M., Weippl, E., Krombholz, K.: User mental models of cryptocurrency systems-a grounded theory approach. In: Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020), pp. 341–358 (2020)
35. McCorry, P., Heilman, E., Miller, A.: Atomically Trading with Roger: gambling on the success of a hardfork. In: CBT 2017: Proceedings of the International Workshop on Cryptocurrencies and Blockchain Technology (2017)
36. McCorry, P., Hicks, A., Meiklejohn, S.: Smart contracts for bribing miners. In: Zohar, A., et al. (eds.) FC 2018. LNCS, vol. 10958, pp. 3–18. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-58820-8_1
37. Meissner, R.: Gnosis community: Gnosis safe contracts - Executor.sol. https://github.com/safe-global/safe-contracts/blob/main/contracts/base/Executor.sol. Accessed 28 May 2022
38. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
39. Natoli, C., Gramoli, V.: The blockchain anomaly. In: 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), pp. 310–317. IEEE (2016)
40. Nayak, K., Kumar, S., Miller, A., Shi, E.: Stubborn mining: generalizing selfish mining and combining with an eclipse attack. In: 1st IEEE European Symposium on Security and Privacy. IEEE (2016)
41. Neu, J., Tas, E.N., Tse, D.: Ebb-and-flow protocols: a resolution of the availability-finality dilemma. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 446–465. IEEE (2021)
42. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 643–673. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_22
43. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. (CSUR) **22**(4), 299–319 (1990)
44. Schwarz-Schilling, C., Neu, J., Monnot, B., Asgaonkar, A., Tas, E.N., Tse, D.: Three attacks on proof-of-stake ethereum. In: International Conference on Financial Cryptography and Data Security (2022)
45. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 478–493. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_30
46. Sompolinsky, Y., Zohar, A.: Bitcoin's Security Model Revisited. arXiv preprint arXiv:1605.09193 (2016)
47. Todd, P.: Op_checklocktimeverify (2014). https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki
48. Tran, M., Choi, I., Moon, G.J., Vu, A.V., Kang, M.S.: A stealthier partitioning attack against bitcoin peer-to-peer network. In: Proceedings of IEEE Symposium on Security and Privacy (IEEE S&P) (2020)

49. Victor, F., Lüders, B.K.: Measuring ethereum-based ERC20 token networks. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 113–129. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32101-7_8
50. Wohrer, M., Zdun, U.: Smart contracts: security patterns in the ethereum ecosystem and solidity. In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 2–8. IEEE (2018)
51. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**(2014), 1–32 (2014)
52. Wu, L., et al.: EthScope: A Transaction-centric Security Analytics Framework to Detect Malicious Smart Contracts on Ethereum. arXiv:2005.08278 (2020). arXiv: 2005.08278
53. Zhang, M., Zhang, X., Zhang, Y., Lin, Z.: {TXSPECTOR}: uncovering attacks in ethereum from transactions. In: 29th {USENIX} Security Symposium ({USENIX} Security 2020), pp. 2775–2792 (2020)
54. Zhang, R., Preneel, B.: Lay down the common metrics: evaluating proof-of-work consensus protocols' security. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE (2019)
55. Zhang, Y., Setty, S., Chen, Q., Zhou, L., Alvisi, L.: Byzantine ordered consensus without byzantine oligarchy. In: 14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 2020), pp. 633–649 (2020)
56. Zhou, L., Qin, K., Torres, C.F., Le, D.V., Gervais, A.: High-frequency trading on decentralized on-chain exchanges. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 428–445. IEEE (2021)