An Overview of Termination in the Ethereum Blockchain

Luca Olivieri[1][0000−0001−8074−8980], Luca Pasetto[2][0000−0003−1036−1718], Luca Negrini[1][0000−0001−9930−8854], and Pietro Ferrara[1][0000−0002−4678−933X]

[1] Ca' Foscari University of Venice, Italy
{name.surname}@unive.it
[2] University of Luxembourg, Luxembourg
luca.pasetto@uni.lu

Abstract. The emergence of the Ethereum blockchain and the rise of Turing-complete smart contracts have led to the creation of new solutions for ensuring different kinds of termination. Indeed, non-termination of a smart-contract execution within the blockchain network may have critical consequences, ranging from slow performance to a complete denial of service in the worst scenarios. Furthermore, smart contracts lack a global access-control mechanism and may be executed indefinitely over time, even after they have exhausted their purposes. Therefore, this requires, in some cases, developing solutions implementing *"soft"* and *"hard"* terminations—such as pausable, interruptions, and *kill-switch* mechanisms—as well as providing *safe* termination guarantees. In addition, termination is even more crucial when we consider legal aspects of smart contracts, including compliance with laws and regulations, such as the smart contract requirements proposed by the *European Union Data Act*. In this paper, we explore several mechanisms to ensure various kinds of termination in Ethereum, the most widely used blockchain. Moreover, we investigate similar mechanisms for traditional programming languages that can be applied to smart contracts in the blockchain context. The primary purpose of this study is to fill the gap caused by the lack of standards for these mechanisms and the emerging solutions typically proposed by practitioners.

Keywords: Smart contract, blockchain, distributed ledger technology, termination, kill switch, interruption, alt, pause, stop, revert, upgradable, undo, rollback, restore, design pattern, legal contracts, EU Data Act

1 Introduction

The blockchain is a distributed ledger that is shared among a decentralized peer-to-peer network, where decisions are made through a consensus mechanism and transactions containing data are recorded and grouped into immutable blocks within the ledger. In 2008, Bitcoin [5, 46] introduced the first killer application implementing a protocol based on the blockchain to exchange economic assets without third-party intermediaries and in a pseudo-anonymous way. Later, in

2014, the Ethereum [6, 65] platform proposed a similar protocol but including the deployment and execution of Turing-complete smart contracts within the blockchain thanks to the Ethereum Virtual Machine (EVM), a decentralized computing virtualized environment for code execution. In particular, Ethereum smart contracts are stateful computer programs written in a Turing-complete low-level language called EVM bytecode. Compilers from high-level languages such as Solidity generate it. Then, bytecode instructions (aka EVM opcodes) are deployed through a transaction into the blockchain, where they are immutably stored. The code execution of smart contracts within the blockchain occurs through transactions that contain execution proposals and are subsequently carried out by the EVM. This new type of blockchain has contributed to the wide diffusion of decentralized applications (DApps), i.e., blockchain-based applications implemented through smart contracts, and has attracted the attention of enterprises, academia, and governments.

Termination of smart contracts is a complex and essential aspect of blockchain systems, carrying different meanings and implications depending on the context, whether technical or legal. From a computer-science perspective, termination refers to the property of a program or code execution halting after a finite number of steps. Ensuring termination is vital to prevent infinite or unbounded computations. On a blockchain, if a piece of code executed by a node fails to terminate, it can cause resource exhaustion, degrade node performance, and increase latency. When such behavior spreads across multiple nodes, it can lead to a denial-of-service (DoS) scenario, potentially disrupting the blockchain's operation or interfering with consensus protocols, which often depend on majority agreement among nodes. In addition, ensuring *safe* termination means that a program not only terminates but does not cause unexpected, malicious, or adverse behavior, such as leaving the system in an inconsistent, vulnerable, or corrupted state.

In a broader legal or operational context, termination can also refer to the intentional disabling or suspension of a smart contract's ability to process its functionalities, either temporarily or permanently. Since smart contracts on public, permissionless blockchains like Ethereum are typically immutable and always accessible, this form of termination plays a critical role in risk management. It enables developers or stakeholders to halt contract execution in response to unauthorized activity, fraudulent use, or the discovery of critical vulnerabilities. Such mechanisms are essential for protecting users, preserving trust, and preventing the continued exploitation of faulty or maliciously crafted contracts. Recently, these aspects have also been accentuated by the final text approval of the European Union Data Act [30] regulation, which mandates essential requirements for data-sharing agreements based on smart contracts, including a *"kill switch"* clause for safe termination and interruption:

> *to ensure that a mechanism exists to terminate the continued execution of transactions and that the smart contract includes internal functions which can reset or instruct the contract to stop or interrupt the operation,*

> *in particular, to avoid future accidental executions* (art. 36, par. 1, EU Data Act [30])

According to EU Crypto Initiative [38], these behaviors can be classified into two different types of termination: *soft termination* and *hard termination*. Soft termination occurs when a program is temporarily paused/interrupted, i.e., an access-control mechanism remains active, and it may be triggered in order to restart the core program operations. Instead, hard termination occurs when the pausing becomes irreversible, and this de facto "terminates" the program because core operations can no longer be performed. Despite the several critical implications of termination, there is currently a lack of comprehensive information on the topic and how to effectively implement termination in blockchain systems. Specifically, there are no established standards to guide best practices, scientific literature on the topic is limited, and most solutions are provided by practitioners who tailor them to specific use cases without verification or broad applicability.

The goal of this paper is to offer a comprehensive understanding of termination mechanisms within the Ethereum ecosystem. In this regard, the paper clarifies the different types of termination, provides examples and code snippets of the current state-of-the-art and state-of-practice, and discusses verification solutions to formally ensure termination occurs correctly.

The key contributions of this paper are the following:

- a comprehensive summary of the different kinds of termination in Ethereum;
- an investigation of the possible solutions to ensure termination and interruptions of code executions;
- an investigation of rollback techniques to restore previous smart-contract states;
- an investigation of techniques to mitigate the consumption of gas and funds incurred for termination due to failures, errors, or unsafe states; and
- an investigation of verification techniques to formally guarantee the termination of smart-contract executions.

Paper structure Section 2 examines the mechanisms adopted by the blockchain to force program termination through the gas mechanism. Section 3 discusses soft and hard terminations, proposing solutions to achieve them at the smart-contract level, along with related pitfalls. Section 4 investigates safe-termination mechanisms. Section 5 discusses related work, and Section 6 concludes the paper.

2 Mechanisms for Program Termination

In the blockchain context, ensuring program termination of smart contracts is crucial to prevent issues like infinite code execution that could lead to resource exhaustion and denial of service. However, determining whether a non-trivial program can terminate for each input is undecidable (the well-known *halting problem* [60]).

The solutions to enforce program termination vary, ranging from time to instruction-count limits. However, among them, the most famous is the *gas mechanism* proposed by Ethereum [6, 65]. When a smart contract is executed, it also sets an amount of gas that it *"burns"* during its execution. If the gas is depleted before the execution is completed, then the contract execution is halted, leading to a transaction failure, and any changes made during the execution are rolled back. Gas units are purchased with cryptocurrency to avoid the abuse of the network. In addition, a maximum limit for consumable gas is set by the Ethereum protocol to avoid large executions by wealthy users that could congest the network. Therefore, the gas system ensures that a program's execution ends regardless of the input. Furthermore, thanks to the rollback in case of failed transactions due to running out of gas, this termination can also be considered safe because it restores a previous state without corrupting or modifying the smart-contract state.

2.1 Pitfalls and Security Implications of Gas Mechanism

Although the gas mechanism is a powerful solution, it also introduces potential security risks, known as out-of-gas vulnerabilities [4, 35], which in 2018 affected Ethereum contracts with a combined value of over $2.8 billion [35]. In these cases, the attack surface includes the maximum gas limit per transaction and components that can be dynamically increased over time, such as collections and arrays. For instance, if a large number of elements are generated, an array might become so long that its manipulation exceeds the maximum gas (metering of code execution) allowed for Ethereum transactions, potentially leading to a denial-of-service of functionalities involving that data structure [25, 26].

3 Mechanisms for Soft and Hard Termination

As reported in Section 1, the concepts of soft and hard terminations for smart contracts originate mainly from domains outside traditional computer science. Furthermore, Ethereum's gas mechanism is designed to enforce economic bounds and ensure program termination, and for this reason is not suitable for these purposes.

Soft termination may evoke the notion of a *"pause"* in traditional software systems, i.e., the act of suspending the rescheduling of a process or task and putting it into an idle state, eventually resuming it once an explicit or implicit event is triggered. However, this analogy cannot fully fit the behaviors of Ethereum smart contracts, because there is no way to truly pause a contract. The code of a smart contract becomes immutable (that is, it cannot be later modified) and it is publicly accessible and available after its deployment in the Ethereum blockchain. This means any Ethereum user can send a transaction request for the smart contract to execute, because there is no global block on contract access. Hence, in this specific context, we can define the notion of *"pause"* as a way implemented in the internal logic of the contract, preferably governed by

some form of access control, for making the program exit (i.e., some code not being executed) under certain circumstances. In other words, this form of pause allows the contract to bypass or disable certain execution paths, thanks to the mutability of the contract's state, which can be used to control execution flow. Similarly, hard termination can be achieved by permanently setting the state to disable any contract's functionality. Additionally, hard termination can also be achieved by a specific native Ethereum instruction to delete contracts.

Below, we discuss the different ways to implement and achieve soft and hard terminations in Ethereum smart contracts. Additionally, we also focus on the security implications and concerns of different approaches.

3.1 Termination through conditional statements

```
1  contract Pausable {
2
3    address public adminAddress;
4
5    constructor() {
6      // store the address of the contract deployer
7      adminAddress = msg.sender;
8    }
9
10   bool public pause;
11
12
13   function transfer(address receiver, uint256 amount) external {
14
15     // pause check
16     require(pause == false, "The transfer is paused");
17
18     // logic for the token transfer
19     require(balances[msg.sender] >= amount, "Not enough tokens");
20     balances[msg.sender] -= amount;
21     balances[receiver] += amount;
22   }
23
24   function setPause(bool _pause) public {
25     require(msg.sender == adminAddress, "Invalid sender");
26     pause = _pause;
27   }
28 }
```

Fig. 1: Pausable implementation based on a boolean flag, which temporarily inhibits the execution of the core token transfer instructions.

As suggested by Wohrer et al. [64], a simple way to implement soft termination into an Ethereum smart contract is by using conditional statements and Boolean guards acting like a switch button to inhibit parts of the code, effectively pausing it. Figure 1 shows a code snippet written in Solidity implementing a simple pausable mechanism. The global variable `pause` declared at line 10 manages the pause capabilities. It can be set by the `setPause` method, and it is exploited within the `transfer` method to drop the code execution, thus inhibiting the transfer of tokens. Specifically, if `pause` is `false`, the `require` statement

```
1 function terminate() public {
2
3   require(msg.sender == adminAddress, "Invalid sender");
4
5   require(pause == false, "Value has already been set");
6   pause = true;
7 }
```

Fig. 2: Write-once mechanism which forces the contract to be permanently paused.

at line 16 does not have any effect on the execution flow. Otherwise, if `pause` is `true`, the `require` statement at line 16 halts the execution returning the message `"The transfer function is paused"` in the transaction response. Note that the `require` statement at line 25 ensures that only the admin user can activate the pause.

Similarly, it is also possible to implement hard termination of the transfer functionality simply by implementing `pause` as a *write-once* variable. To do this, we simply need to implement the same contract in Figure 1 by replacing the `setPause` method with the `terminate` method proposed in Figure 2. In particular, in this way, it is no longer possible to change the value of `pause` arbitrarily since the `setPause` method is missing. When the contract is deployed for the first time, the value of `pause` will be the default one, i.e., `false`. The only way to change it is through `terminate`, which, after the first execution, leads to an irreversible contract state because it sets `pause` to `true`, and the value of `pause` cannot be changed anymore.

For the sake of simplicity, Figures 1 and 2 assume that the user who deploys the contract is the admin user. The assignment is implemented in the constructor at line 7, and the value of admin's address is stored in field `adminAddress` at line 3. Note that the visibility of `adminAddress` is `public`, meaning it can be read externally outside the contract (e.g., through web APIs such as web3.js or by other contracts), but can only be modified from inside the contract unless there is an explicit setter function provided. Then, in the case of Figure 1, it can be set only by invoking the constructor. In state-of-the-art implementations, the most popular pausable contracts, such as those proposed by OpenZeppelin [53, 55], implement more sophisticated access control layers to handle who and how a user or the contract can pause operations. Instead, our contract focuses on the termination problem, and more sophisticated access control mechanisms can be added without modifying the termination functionalities.

3.2 Termination through cross-contract invocations

In Ethereum, it is possible to execute cross-contract invocations between smart contracts already deployed in the blockchain. This means that a smart contract can invoke a function in another deployed smart contract through the `CALL`, `STATICCALL`, and `DELEGATECALL` opcodes. Cross-contract invocations can be

exploited to communicate with other contracts for data exchange (e.g. values, assets, and cryptocurrencies), to build libraries of shared code that multiple contracts can access, to overcome limitations related to contract size [17], and to circumvent code immutability through proxy upgrade patterns [45] to include new features and patches in DApps. They can also implement soft and hard termination mechanisms. For instance, the proxy upgrade pattern [45] allows the building of a DApp based on multiple smart contracts where it is possible to upgrade the application logic while maintaining the same contract address. Then, in the settings of soft and hard terminations, it is possible to implement a "switch" to inhibit parts of code by changing the target of cross-contract invocations.

Considering the simplified version of a proxy upgrade pattern scenario in Figure 3, where the code proxy contract with address `0xa...` is reported in Figure 4 and the logic contracts with addresses `0xb...` and `0xc...` are reported in Figure 5a and Figure 5b, respectively. The proxy upgrade pattern can be viewed as a *template* [32] design pattern where a *proxy contract* acts as a *template* providing *hook functions* that perform cross-contract invocations to other deployed contracts containing the application logic. In Figure 4, the hook function is `foo` at line 20, while the concrete functions that implement the application logic are the `foo` functions at lines 10 and 4 in Figure 5a and Figure 5b, respectively. In these cases, the cross-contract invocations are parametric. The parameter values are stored in the contract state, and they can typically be modified by user input. This proxy maintains the contract state, and it can be updated to point to a new implementation, allowing the underlying application logic to be upgraded without changing the proxy contract's address. In this way, it is possible to inhibit functionalities by redirecting cross-contract invocation to a no-op function of the same or another logic contract. In Figure 4, the cross-contract invocations are executed by the `delegatecall` at line 25 which calls a hardcoded function defined as `"foo(uint256)"` from a contract that is specified by the address coming from in the global variable `logicContract`, which can be changed over the time by the admin user using the method `upgrade` at line 13. Note that, it is a delegate call because it allows the proxy to execute the logic contract's code in its own context. In the settings of Figure 3, we have the concrete application logic at the address `0xb...`, i.e. the `foo` method of Figure 5a that increase a counter, while the `foo` method of Figure 5b at the address `0xc...` correspond to the no-op. Then, it is possible to turn on and off the application logic, switching the addresses `0xb...` and `0xc...` respectively by playing with the `upgrade` method of Figure 4.

Compared with the termination through conditional statements only, the solution with cross-contract invocations is generally economically more expensive because it requires one to consume a greater amount of gas to deploy multiple smart contracts. Indeed, as currently reported by the fee schedule in the Yellow Paper of Ethereum [65, Appendix G], the cost paid for each contract-creating transaction is at least 32000 units of gas against the few dozen units consumed by each basic EVM opcode that makes up the conditional statements. Moreover, the

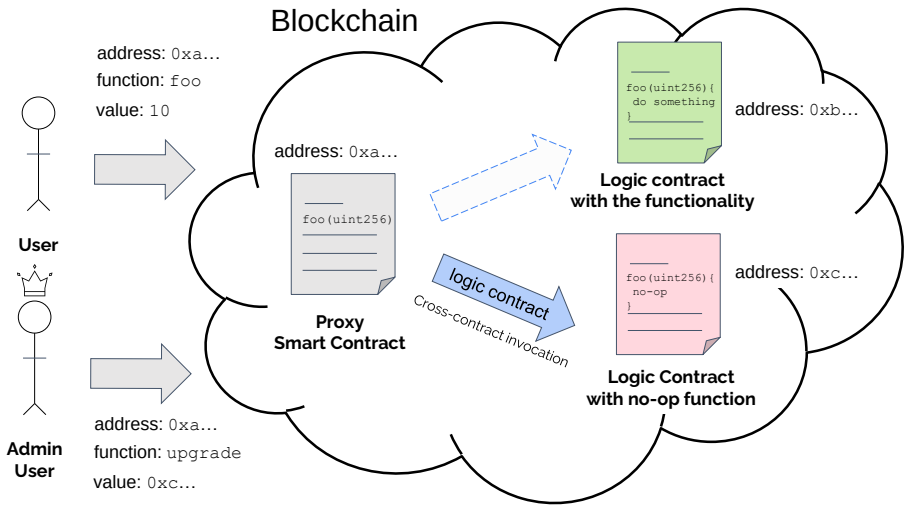

Fig. 3: Termination scenario with upgrade pattern

implementation of upgradable contracts is error-prone due to non-trivial security implications such as correct management checks, restrictions, or versioning [42]. According to Huang et al. [37], it is also necessary to consider transparency mechanisms because a contract may be quietly upgraded, when the functionality of the contract has changed, without the user being aware that the contract has been upgraded, which may lead to financial losses for the users. Furthermore, in the worst-case scenario, if the contract runs a malicious update, there are greater risks of a user becoming a victim. However, termination through cross-contractual invocations allows the evolution of the DApp, with the possibility of implementing new switches over time, allowing for greater possibility of future extensions.

3.3 An instruction for the Self-destruction

As mentioned previously, the code of smart contracts is immutable and cannot be changed. However, the EVM opcode `SELFDESTRUCT` (previously called `SUICIDE` [7]) with its execution allows one to *"delete"* a contract and send any contract funds to a specified address [6], ensuring hard termination.

In practice, during a smart contract deployment in Ethereum, the code is uniquely associated to a special type of account called *smart contract account*, which will be controlled by the code rather than private keys as it happens instead for user accounts. In this way, the code can be identified in the network thanks to the account address. The semantics of `SELFDESTRUCT` removes the code and its internal state (storage) from the contract address, leaving a blank account. Furthermore, if a transaction with an execution proposal is sent to an empty account, no code execution results, since there is no longer any code to

```
contract Proxy {

  address public adminAddress;

  constructor() {
    // store the address of the contract deployer
    adminAddress = msg.sender;
  }

  address public logic contract;

  // set of the logic contract after deployment
  function upgrade(address _newlogicContract) public {
    require(msg.sender == adminAddress, "Invalid sender");

    logicContract = _newlogicContract;
  }

  // hook function
  foo(uint256 _value) external payable {
    address logic = logicContract;
    require(logic != address(0), "Logic contract not set");

    //  cross-contract invocation
    (bool success, bytes memory data) = logic.delegatecall(abi.encodeWithSignature("foo(
        uint256)", _value));
    require(success, "Delegatecall failed");
  }
}
```

Fig. 4: Proxy contract based on external calls that allows the admin to change the target contract containing the application logic.

execute there. However, note that the `SELFDESTRUCT` only affects the state of the account. Therefore, it does not remove the transactions' history of the contract prior to the `SELFDESTRUCT` execution since the blockchain itself is immutable. Furthermore, it is also important to note that `SELFDESTRUCT` must be present in the contract code and be executed to have its effect. Indeed, if the contract code does not have a `SELFDESTRUCT` opcode (not included by default) or this is not reachable during the execution, the smart contract cannot be deleted.

The `SELFDESTRUCT` instruction has always been highly debated in the Ethereum community due to security and trust concerns. According to Chen et al. [20], `SELFDESTRUCT`, this is a double-edged sword for developers. On the one hand, it enables contract owners to have the ability to reduce financial loss when emergency situations happen or when the code has ceased to serve its purposes. On the other hand, this function is also harmful because it opens attack vectors for malicious users. Moreover, according to Buterin [18], `SELFDESTRUCT` is the only EVM opcode that breaks important invariants: (i) it causes an unbounded number of state objects to be altered in a single block, (ii) it cause the code of a contract to change, and (iii) it can change other accounts' balances without their consent. For these reasons, in 2022, the EVM opcode `SELFDESTRUCT` has been proposed for the deprecation [29], and over time, several alternatives have been proposed for its replacement or to change its semantics (e.g., [8, 9, 12]).

```
contract Logic1 {
  address constant public proxyAddress = 0
       xa...;

  uint256 public counter;

  constructor() {
    counter = 0;
  }

  function foo(uint256 _value) public {
    // drop execution if the sender is not
         the proxy
    require(msg.sender == proxyAddress, "
         Invalid sender");
    // application logic
    counter = counter + _value;
  }
}
```

(a) Enabled function

```
contract Logic2 {
  address constant public proxyAddress = 0
       xa...;

  function foo(uint256 _value) public {
    // drop execution if the sender is not
         the proxy
    require(msg.sender == proxyAddress, "
         Invalid sender");

    // no-op
    require(false, "This function is
         disabled")
  }
}
```

(b) Disabled function

Fig. 5: Target contracts which enable and disable the application logic of proxy contract.

3.4 Pitfalls and Security Implications of Hard Termination

According to Ezeozue [39], changes and actions to `SELFDESTRUCT` may reduces certain attack vectors, but significant risks persist. For instance, EIP-6780 [9] in 2023 modified `SELFDESTRUCT` semantics to only clear contract code and storage if executed in the same transaction as contract creation. However, this creates compatibility issues in terms of blockchain interoperability. Indeed, some major EVM-compatible blockchains (e.g. Binance Smart Chain (BSC) and certain Polygon chains) have not yet enforced EIP-6780 restrictions, leaving `SELFDESTRUCT` fully functional. Furthermore, this still leaves unresolved issues related to proxy, implementations [27, 39], such as front-run initialization transactions to execute `SELFDESTRUCT` before proper setup is completed.

Regarding hard termination vulnerabilities, they are mainly caused by insufficient and/or improperly implemented access control, which can unexpectedly allow users to access hard termination functions [59]. According to Ressi et al. [59], those related to `SELFDESTRUCT` can be referred in different ways in literature. The terms *Guard Suicide* [19], *Unprotected Suicide* [36, 44], *Suicidal Contract* [47], *Destroyable contracts* [15] or simply *Suicide* [31] identify cases where an attacker deliberately destroys smart contracts, and eventually performing an token transfers to specific smart contracts that were not supposed to receive them, i.e. those containing a `SELFDESTRUCT`. Specifically, according to Brent et al. [14], they can be classified in two different types: (i) *accessible self-destruct*s that can allow any user to trigger the contract destruction and (ii) *tainted self-destruct*s where an attacker can also control/become the receiver address of the funds returned by the contract destruction. Furthermore, the notion of *accessible self-destruct* can be also generalized in *accessible hard termination* including also all operations,

not only `SELFDESTRUCT` operations, that leads to a permanent and irreverable paused state.

4 Safe Termination

The term *safe termination* can have different meanings depending on the context in which it is used, and can be understood both in terms of preventing errors or failures and as protection against malicious actors.

In the blockchain context, a contract can be terminated due to an unexpected error, data corruption, or malicious activity by an administrator. In these cases, termination should guarantee that the contract remains in a safe state where no data is corrupted and malicious activities are prevented. For example, this could involve denying transactions that could lead to an insecure termination, restoring a previously safe state, and, if possible, returning any spent funds or gas.

In general, this is often unfeasible because it would require rolling back transactions already stored immutably on the blockchain. In fact, data changes at the blockchain level are extremely rare events in permissionless blockchains, since the absence of a central entity requires that the majority of the blockchain network, often made up of several thousand peers, must agree. Furthermore, this could lead to a *hard fork* in which the blockchain is split in two, and some nodes choose the restored version while others keep the original one without rollback. For instance, a hard fork due to a transaction rollback in the Ethereum blockchain occurred because of the *DAO attack*, which affected the majority of peers and users, causing economic damage exceeding $50 million [57].

Although transactions are immutable within the blockchain and cannot be modified, it is still possible to modify the state of smart contracts and mitigate gas consumption in the event of errors or failures. In the rest of this section, we discuss revert and undo mechanisms that can partially address state-restore issues, as well as methods to mitigate gas consumption using specific Ethereum instructions at the smart-contract level.

4.1 Restore through Snapshots

Although rolling back transactions already stored in the blockchain is almost impracticable, this does not exclude the possibility of acting on the smart contract state to totally or partially restore a previously safe state.

In current practice, some Ethereum smart contracts for tokens implement mechanisms to collect snapshots of their states over time [54]. According to Crosara et al. [25,26], the main purpose of snapshots is to provide an immutable view of the ledger that a client can query without the risk that it changes during the query, which would result in a race condition. Furthermore, they are useful for investigating the consequences of an attack, for creating forks of the token and for implementing mechanisms based on token balances such as weighted

voting. In practice, these contracts implement a mapping between addresses and snapshot structures containing the values to be remembered.

Similarly, it is possible to design a solution that exploits a snapshot-like structure to collect a smart contract state and then restore its information over time. In traditional software programming, the *memento* [32,58] design pattern is particularly involved in scenarios where one needs to implement undo or rollback functionalities. It can be used to capture and store the current state of an object so that it can be restored later without breaking the encapsulation.

Considering the rollback scenario based on the memento pattern proposed in Figure 6, the code of the originator contract is reported in Figure 7, the code of the caretaker in Figure 8, and the code of memento instances in Figure 9.

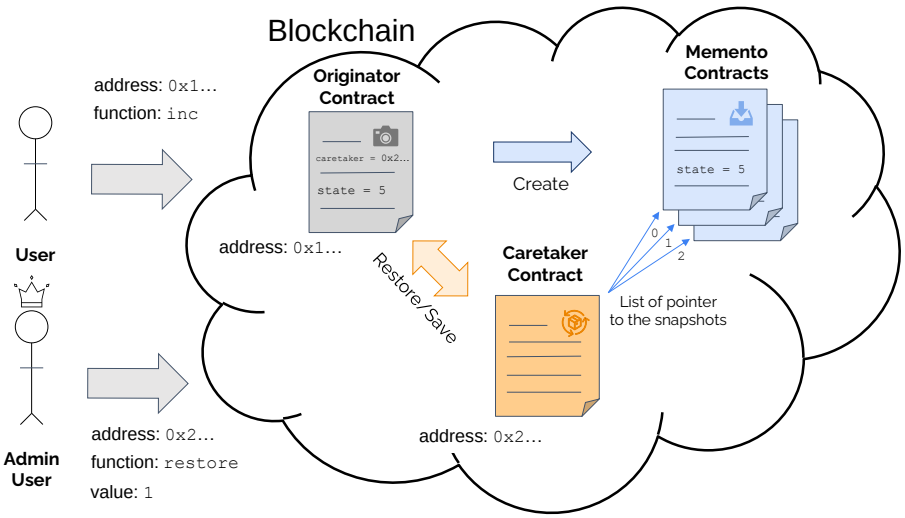

Fig. 6: Rollback scenario with memento pattern

The purpose of the originator contract is to handle application logic of the state that one wants to save and restore. For simplicity, in Figure 7, the state is just an integer value collected in the variable `state` at line 7, and the application logic is the increment function `inc` at lines 14-16 that can be called by to change the variable `state`. The operations of save and restore are instead delegated to the caretaker.

As reported in Figure 8 at lines 11-14, an originator contract is created, and an admin is set when a caretaker contract is deployed. In this way, an admin can save the states of originator by calling the function `saveState`. This function at line 18 asks the originator to create a snapshot of its state (Figure 7 at lines 18-21), i.e., a memento contract. It then collects the snapshot in an array at line 19. These collected states can be restored by the admin calling the function

```
1  import "./Memento.sol"
2
3  contract Originator {
4
5    address public caretakerAddress;
6
7    uint256 public state;
8
9    constructor(address _caretakerAddress) {
10     caretakerAddress = _caretakerAddress;
11     state = 0;
12   }
13
14   function inc() public {
15     state = state + 1;
16   }
17
18   function createMemento() public view returns (Memento) {
19     require(msg.sender == caretakerAddress, "Invalid caretaker address");
20     return new Memento(state);
21   }
22
23   function restoreMemento(Memento memento) public {
24     require(msg.sender == caretakerAddress, "Invalid caretaker address");
25     state = memento.getState();
26   }
27 }
```

Fig. 7: Originator contract which contains the application state and allows the caretaker to save it or restore a previous one.

`restoreState` that at line 25 asks the originator to replace the current state value with that of the selected snapshot (Figure 7 at lines 23-26).

As shown in Figure 9, the memento contract collects just the values of the originator contract at lines 4-6 in variable `state`, and it provides the function `getState` at lines 8-10 to return to the saved state.

Alternatively, it is possible to manage snapshots not as contracts but as structures within the Caretaker contract using the `struct` data type and declaring custom fields. However, the advantage of contracts is that they can also embed code and utility functions.

The main limitation of this approach is that it is not possible to fully restore all the values of a contract state from a previous snapshot. When a transfer token or funds are involved, the moved tokens and funds cannot be forced back to their owners because transfers are irreversible actions without a fork.

4.2 Lock of resources

While it is not possible to roll back a token transfer without performing a fork, techniques to mitigate such behavior can still be applied. Lockup mechanisms in smart contracts refer to features that restrict the movement or transfer of resources, assets, or tokens for a specified period. These mechanisms are commonly used in decentralized finance (DeFi) protocols and cross-chain token transfer protocols to enhance security, prevent premature selling, and align incentives among participants.

```
import "./Originator.sol";
import "./Memento.sol";

contract Caretaker {

  address public adminAddress;

  Originator private originator;
  Memento[] private mementos;

  constructor() {
    adminAddress = msg.sender;
    originator = new Originator(address(this));
  }

  function saveState() public {
    require(msg.sender == adminAddress, "Invalid admin address");
    Memento memento = originator.createMemento();
    mementos.push(memento);
  }

  function restoreState(uint256 index) public {
    require(msg.sender == adminAddress, "Invalid admin address");
    require(index < mementos.length, "Invalid index");
    originator.restoreMemento(mementos[index]);
  }
}
```

Fig. 8: Caretaker contract that orchestrates the originator contract and all its snapshots stored as memento contracts.

In the context of safe termination, a lockup mechanism can allow a fund transfer to be postponed and kept pending for a specified period. This provides an opportunity for an admin user to intervene in case of anomalies and return the resources to the original owner before contract termination.

Figure 10 implements a smart contract with temporized lockup logic to handle fund transfers in Solidity. The idea is that a sender user can submit transfer proposals to the contract calling the function `transferProposal` at lines 19-35, specifying the recipient and the duration of the time window in which the admin can roll back the transaction. Then, the time window is computed at line 23, and transaction information, including the value to transfer, is collected and locked in a map at lines 25-32. At this point, the recipient can claim the fund transfer from the sender if the time window has expired by calling the function `claimTransfer` at lines 37-47, which will retrieve the transfer information from the map containing the transfer proposals and will perform the transfer if the proposal has not been reversed previously during the time window.

The revert can be performed only by the admin user calling the function `revertTransfer` at lines 49-62 that gets the transfer proposal and send back the funds to the sender at line 61 if the time windows are still valid at line 58.

For simplicity, the code does not allow you to notify the administrator via code. However, it can be implemented by enriching the program with *events* and using the `emit` instruction to notify the admin client.

The main drawbacks of this solution are reduced decentralization, potential for abuse, and increased operational complexity. Additionally, the transfer would

```
contract Memento {
  uint256 private state;

  constructor(uint256 _state) {
    state = _state;
  }

  function getState() public view returns (uint256) {
    return state;
  }
}
```

Fig. 9: Memento contract that captures a snapshot of the originator's state at a specific point in time.

still be carried out if the administrators were not notified in time during the time window or were unable to carry out the operations.

4.3 Reverting Transactions and Fee Recovery

In Ethereum, transaction failures during smart-contract execution are mainly due to running out of gas or executing an invalid instruction. In these cases, a reversal of the operations occurs, and all the transaction gas is consumed. Once this happens, the gas is deducted from the sender's balance and is paid to network peers involved in processing the transactions. Moreover, the revert of EVM execution means that all changes, including log information, are lost and there is no way to convey a reason for aborting an EVM execution.

To avoid this scenario, it is possible to add to a smart contract the `REVERT` [13] opcode that provides a way to stop execution and revert state changes, consuming only the gas used up to that point, preventing all the transaction gas from being consumed. Additionally, the `REVERT` instruction also provides a pointer to a memory section, which can be interpreted as an error code or message.

Note that transaction fees will still be burned, and no log information will be saved if the contract code does not include a `REVERT` instruction (which is not included by default), the instruction is unreachable during execution, or the cases mentioned above occur before `REVERT` is executed.

The `REVERT` instruction is typically used for error handling, assertions, and custom revert reasons based on the application logic.

Specifically, in Solidity, there are few high level instructions that include `REVERT` for these purposes. For instance, `require()` validates conditions and refunds unused gas upon failure. While, `revert()` provides a flexible alternative to `require()` encouraging the use of custom errors over string messages and improving gas efficiency and code clarity [63]. By contrast, `assert()` checks invariants but does not refund unused gas, because its compilation omits `REVERT`. For this reason, developers typically advise against overusing `assert()` to prevent excessive gas usage.

```
contract TemporizedFundTransfer {

  address public adminAddress;

  struct Transfer {
    address sender;   address recipient;
    uint256 amount;   uint256 releaseTime;
    bool claimed;     bool reverted;
  }

  uint256 transferId;
  mapping(uint256 => Transfer) public transfers;

  constructor() {
    adminAddress = msg.sender;
    transferId = 0;
  }

  function transferProposal(address recipient, uint256 duration) external payable {
    require(msg.value > 0, "Must send some funds");
    require(recipient != address(0), "Invalid recipient");

    uint256 releaseTime = block.timestamp + duration;

    transfers[transferId] = Transfer({
      sender: msg.sender,
      recipient: recipient,
      amount: msg.value,
      releaseTime: releaseTime,
      claimed: false,
      reverted: false
    });

    transferId = transferId + 1;
  }

  function claimTransfer(uint256 transferId) external {
    Transfer storage transfer = transfers[transferId];

    require(msg.sender == transfer.recipient, "Invalid recipient address");
    require(block.timestamp >= transfer.releaseTime, "Transfer is still locked");
    require(!transfer.reverted, "Transfer already reverted");

    require(!transfer.claimed, "Transfer already claimed");
    transfer.claimed = true;
    payable(transfer.recipient).transfer(transfer.amount);
  }

  function revertTransfer(uint256 transferId) external {

    // drop execution if the sender is not the admin
    require(msg.sender == adminAddress, "Invalid sender");

    Transfer storage transfer = transfers[transferId];

    require(!transfer.claimed, "Transfer already claimed");
    require(!transfer.reverted, "Transfer already reverted");
    require(block.timestamp < transfer.releaseTime, "Lockup period has ended");

    transfer.reverted = true;
    payable(transfer.sender).transfer(transfer.amount);
  }
}
```

Fig. 10: Temporized transfer lockup mechanism with a revert capability.

5 Related Work

Despite the importance of topics related to termination, the current literature in the blockchain context offers only limited results, with few studies providing detailed examinations. Compared to this article, related work tends to focus narrowly on specific aspects of termination, often presenting only partial overviews and omitting implementation details, or failing to address the associated issues and pitfalls comprehensively.

For the sake of readability, the literature can be divided into two strands: (i) verification methods for proving and ensuring termination and (ii) the implications of termination in the legal sphere.

5.1 Literature about Termination Verification

Ensuring smart contract termination is necessary in several contexts. However, the only way to prove that a program terminates is through formal methods.

In general, program termination analysis has benefited from many research advances, and several tools have emerged over the years [16, 22, 28, 34, 40]. According to Courant and Urban [23], traditional methods for proving program termination rely on the synthesis of a ranking function, a well-founded metric that strictly decreases during program execution, which quantifies the remaining distance to termination. Moreover, it is also possible to exploit abstract interpretation [56] to approximate the most precise ranking function [23, 24, 62].

On the other hand, for the blockchain context, Le et al. [41] describe a preliminary study for a static lazy approach to proving conditional termination and non-termination of a smart contract by determining the input conditions under which the contract terminates or not and whether the contract is qualified (i.e., eventually terminating) to run on the blockchain. In particular, they consider non-termination due to infinite loops in smart contracts of blockchain without a gas mechanism, such as Hyperledger Fabric. In contrast, they consider termination failure due to insufficient gas for blockchains with a gas mechanism, such as Ethereum. In this approach, when a smart contract is submitted to the blockchain, the system first automatically calculates logical formulas that determine the preconditions of the contract's inputs and the chain states under which its execution terminates or not. These formulas are verifiable and then recorded into the blockchain as metadata of the contract. Later, when a transaction invokes the contract, the system will check if the current blockchain state and the contract's input satisfy any recorded termination precondition. If this is the case, then the contract is executed. Otherwise, the transaction is aborted. However, according to Olivieri et al. [51, 52], the assumption of applying the verification directly on the blockchain may have non-trivial impacts on the system, such as slowdowns or performance drops.

However, unlike traditional software, according to Genet et al. [33], reasoning about the gas mechanism in blockchain software could make program termination of contracts easier to prove, i.e., proving that *"it is impossible to construct an infinite loop that does not consume any gas"*. However, the official definition

of gas usage makes the proof of this property complex due to the decidedly non-trivial semantics of contract calls and the fact that cash-in of call cost is delayed until after the return in both regular and exceptional cases. Furthermore, it is also necessary that the gas model and the implementation are sound, i.e., during the execution, the gas must be burned correctly to avoid a greater waste of resources or, in the worst case, non-termination issues. More technically, Genet et al. [33] propose a formal and general proof of termination of smart contracts based on a measure of EVM call stacks. They proved that no program can execute indefinitely without consuming gas in the EVM execution model by leveraging the Isabelle/HOL proof assistant to mechanize the proof. The model is sound by leveraging safe over-approximates of the EVM semantics with minimal assumptions on the concrete gas costs due to the fact that the costs has already changed several times during the life of the EVM.

There are also studies to verify the detection of unexpected terminations due to out-of-gas issues related to the gas limit caps in the smart contract execution. Grech et al. [35] present a tool to detect gas-focused vulnerabilities in Ethereum smart contracts automatically. It performs a static analysis that combines abstract-interpretation-based low-level analysis for decompilation of EVM bytecode, and declarative program analysis techniques for higher-level analysis. The main limitation of the tool is that it provides a *soundy* implementation, i.e., it does not provide a guarantee of identifying all gas vulnerabilities, nor that the reported vulnerability is a real bug. The reasons for this implementation choice are due to the need to scale to a very large number of contracts. A sound gas analyzer is instead proposed by Albert et al. [4]. It automatically infers upper bounds on the gas consumption for each public function of Ethereum smart contracts by relying on existing cost analysis techniques [2,3]. Then, the tool allows one to identify functions with a constant memory gas consumption and functions with a memory gas bound that is not constant, which could lead to out-of-gas vulnerabilities.

Soft and hard termination, on the other hand, require verifying the code of smart contracts. However, according to Olivieri et al. [51], this kind of verification is challenging due to cross-component interactions of multiple contracts and the need for formalization of the properties to be proven. Indeed, in DApps based on multiple contracts, it may require that multiple parts of code are inhibited and in specific orders to prevent the individual components from being used alone and improperly. Furthermore, the verification becomes much more complex if one also want to consider cross-chain and multi-chain scenarios [48].

Moreover, current smart contract verification tools are not primarily focused on ensuring soft and hard termination. However, this challenge might be addressed similarly to how the liquidity property is verified [11]. According to Bartoletti et al. [10], ensuring the liquidity property means that from *"every reachable state a user can execute a sequence of transactions to withdraw a given amount of crypto-assets"*, reducing the verification problem to symbolic model checking.

In a similar manner, soft and hard termination can be framed as ensuring that, from every reachable state, an admin user can execute a sequence of transactions to terminate the contract. This would reduce the verification problem to model checking [21].

Regarding the detection of vulnerabilities related to hard termination, several works [15, 19, 31, 47] rely on static and dynamic symbolic execution techniques to assess the feasibility of execution paths that include `SELFDESTRUCT` operations without proper guarding conditions, commonly referred to as *accessible self-destruct* [14]. However, these approaches do not analyze the target address of the `SELFDESTRUCT` instruction, and thus cannot detect cases where the target is tainted by user input, known as *tainted self-destruct* [14]. While, Brent et al. [14] applies an information flow analysis tracking tainted data to detect both these types of issues. Ressi et al. [59] investigate detection via machine learning techniques, and Hu et al. [36] apply knowledge-graphs checks. Instead, Mavridou et al. [44] propose a framework for the secure generation of smart contracts using formal methods, ensuring that `SELFDESTRUCT` operations are properly guarded by construction.

5.2 Literature about Termination in the Legal Sphere

Regarding the intersection of computer science and legal aspects, Marino et al. [43] describe how to design alter and undo features based on legal definitions, such as *"Termination by Right"*, *"Rescission by Agreement"*, and *"Rescission by Court"*. They also develop prototypes of Ethereum smart contracts written in Solidity that implement and align with these definitions. In these prototypes, the termination mechanisms are implemented using conditional statements. Olivieri et al. [49, 50] investigate the compliance of blockchain-based smart contracts with the European Union Data Act. Among various compliance issues, they also address termination problems, emphasizing a lack of standards beyond just the "kill switch" clause. However, they do not propose any code implementations in this context.

Instead, Seneviratne [61] provides a study fully focused on the "kill switch" mechanism reported in the European Union Data Act. Compared to our work, it does not go into deep details related to the Ethereum blockchain and does not include technical code details about the smart contracts. However, it provides a high-level overview regarding the mechanisms for smart contract termination across several major blockchains and distributed ledgers (e.g., Ethereum, Cardano, Hyperledger Fabric, IOTA, ...). Abdrashitov et al. [1] discuss the normative regulation and practical aspects of smart contract termination in the investment context, specifically focusing on Russian legislation. They define smart contract termination as the legal termination of an agreement and highlight the inadequacy of Russian regulations, which currently makes it impossible to fully apply the smart contracts to the traditional civil law rules.

6 Conclusion

In this paper, we investigated the termination of smart contract execution on the Ethereum blockchain platform. In general, program termination is ensured by the gas mechanism, which halts smart-contract execution when the gas associated with a transaction runs out due to the execution of smart contract instructions.

Regarding soft and hard termination, solutions can be developed and implemented in the code of smart contracts. These solutions can include kill switches ranging from simple conditional statements to more complex designs inspired by traditional design patterns. However, such solutions generally require access-control policies to enable or disable the kill-switch functionality. This reliance on admin users necessitates trusting a third party, which may not serve users' interests or could arbitrarily block valid transfers. Moreover, this runs counter to the "permissionless" principle, which aims to eliminate third-party intermediaries. Additionally, verification techniques must be applied to ensure the correctness of access-control implementations and the proper activation of soft and hard termination mechanisms.

Finally, achieving safe termination in the blockchain context is challenging because rolling back approved transactions typically requires proposing a fork and securing the consensus of the majority of the blockchain network. However, some program behaviors can be mitigated at the code level to reduce costs in the event of failures, restore snapshots of specific smart-contract states, and add timed lock-up mechanisms for fund transfers.

Acknowledgements

Work partially supported by SERICS (PE00000014 - CUP H73C2200089001), iNEST (ECS00000043 – CUP H43C22000540006) projects funded by PNRR NextGeneration EU, and by the Luxembourg National Research Fund (FNR) (INTER/DFG/23/17415164/LODEX).

References

1. Abdrashitov, V.M., Davudov, D.A., Kolosov, N.F., Slezhenkov, V.V.: Risks of Smart Contracts Termination in the Investment Sphere, pp. 291–298. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-51536-1_27, https://doi.org/10.1007/978-3-031-51536-1_27
2. Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Puebla, G., Román-Díez, G.: Object-sensitive cost analysis for concurrent objects. Software Testing, Verification and Reliability **25**(3), 218–271 (2015). https://doi.org/10.1002/stvr.1569, https://doi.org/10.1002/stvr.1569
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012). https://doi.org/10.1016/j.tcs.2011.07.009, https://doi.org/10.1016/j.tcs.2011.07.009, quantitative Aspects of Programming Languages (QAPL 2010)

4. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: Don't run on fumes—parametric gas bounds for smart contracts. Journal of Systems and Software **176**, 110923 (2021). https://doi.org/https://doi.org/10.1016/j.jss.2021.110923
5. Antonopoulos, A.M.: Mastering Bitcoin: Programming the Open Blockchain. O'Reilly, Sebastopol, CA, USA, 2nd edn. (2017)
6. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly, Sebastopol, CA, USA (2018)
7. Ballet, G., Buterin, V., Feist, D.: EIP-6: Renaming SUICIDE opcode (2015), ethereum Improvement Proposals, no. 6, November 2015. [Online serial]. Available: `https://eips.ethereum.org/EIPS/eip-6` (Accessed 08/2024)
8. Ballet, G., Buterin, V., Feist, D.: EIP-4758: Deactivate SELFDESTRUCT [DRAFT] (2022), ethereum Improvement Proposals, no. 4758, February 2022. [Online serial]. Available: `https://eips.ethereum.org/EIPS/eip-4758` (Accessed 08/2024)
9. Ballet, G., Buterin, V., Feist, D.: EIP-6780: SELFDESTRUCT only in same transaction (2022), `https://eips.ethereum.org/EIPS/eip-6780` (Accessed 08/2024)
10. Bartoletti, M., Ferrando, A., Lipparini, E., Malvone, V.: Solvent: liquidity verification of smart contracts. arXiv preprint arXiv:2404.17864 (2024)
11. Bartoletti, M., Zunino, R.: Verifying liquidity of bitcoin contracts. In: Nielson, F., Sands, D. (eds.) Principles of Security and Trust. pp. 222–247. Springer International Publishing, Cham (2019)
12. Beregszaszi, A.: EIP-6046: Replace SELFDESTRUCT with DEACTIVATE [DRAFT] (2022), ethereum Improvement Proposals, no. 6046, November 2022. [Online serial]. Available: `https://eips.ethereum.org/EIPS/eip-6046` (Accessed 08/2024)
13. Beregszaszi, A., Mushegian, N.: EIP-140: REVERT instruction (2017), ethereum Improvement Proposals, no. 140, February 2017. [Online serial]. Available: `https://eips.ethereum.org/EIPS/eip-140` (Accessed 08/2024)
14. Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 454–469. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3385990, `https://doi.org/10.1145/3385412.3385990`
15. Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B.: Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981 (2018)
16. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 413–429. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
17. Buterin, V.: EIP-170: Contract code size limit (2016), ethereum Improvement Proposals, no. 170, November 2016. [Online serial]. Available: `https://eips.ethereum.org/EIPS/eip-170` (Accessed 08/2024)
18. Buterin, V.: Pragmatic destruction of SELFDESTRUCT (2024), `https://hackmd.io/@vbuterin/selfdestruct\#Pragmatic-destruction-of-SELFDESTRUCT` (Accessed 08/2024)
19. Chang, J., Gao, B., Xiao, H., Sun, J., Cai, Y., Yang, Z.: scompile: Critical path identification and analysis for smart contracts. In: Formal Methods and Software Engineering: 21st International Conference on Formal Engineering Methods, ICFEM

2019, Shenzhen, China, November 5–9, 2019, Proceedings. p. 286–304. Springer-Verlag, Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-030-32409-4_18, https://doi.org/10.1007/978-3-030-32409-4_18

20. Chen, J., Xia, X., Lo, D., Grundy, J.: Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. ACM Trans. Softw. Eng. Methodol. **31**(2) (dec 2021). https://doi.org/10.1145/3488245, https://doi.org/10.1145/3488245
21. Clarke, E.M.: Model checking. In: Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17. pp. 54–56. Springer (1997)
22. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification. pp. 415–418. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
23. Courant, N., Urban, C.: Precise widening operators for proving termination by abstract interpretation. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 136–152. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
24. Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 245–258. POPL '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2103656.2103687, https://doi.org/10.1145/2103656.2103687
25. Crosara, M., Olivieri, L., Spoto, F., Tagliaferro, F.: Re-engineering erc-20 smart contracts with efficient snapshots for the java virtual machine. In: 2021 Third International Conference on Blockchain Computing and Applications (BCCA). pp. 187–194 (2021). https://doi.org/10.1109/BCCA53669.2021.9657047
26. Crosara, M., Olivieri, L., Spoto, F., Tagliaferro, F.: Fungible and non-fungible tokens with snapshots in java. Cluster Computing **26**(5), 2701–2718 (2023). https://doi.org/10.1007/s10586-022-03756-3, https://doi.org/10.1007/s10586-022-03756-3
27. David, E.C.: The Hidden Dangers of Using Selfdestruct in Upgradable Smart Contracts (2025), https://coinsbench.com/the-hidden-dangers-of-using-selfdestruct-in-upgradable-smart-contracts-832466bf6b95 (Accessed 07/2025)
28. D'Silva, V., Urban, C.: Conflict-driven conditional termination. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 271–286. Springer International Publishing, Cham (2015)
29. Entriken, W.: EIP-6049: Deprecate selfdestruct (2022), ethereum Improvement Proposals, no. 6049, November 2022. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-6046 (Accessed 08/2024)
30. European Parliament and the Council: Regulation (EU) 2023/2854 of the European Parliament and of the Council of 13 December 2023 on harmonised rules on fair access to and use of data and amending Regulation (EU) 2017/2394 and Directive (EU) 2020/1828 (Data Act) (2023), document 32023R2854. PE/49/2023/REV/1 OJ L, 2023/2854, 22.12.2023, ELI: http://data.europa.eu/eli/reg/2023/2854/oj
31. Fu, M., Wu, L., Hong, Z., Zhu, F., Sun, H., Feng, W.: A critical-path-coverage-based vulnerability detection method for smart contracts. IEEE Access **7**, 147327–147344 (2019). https://doi.org/10.1109/ACCESS.2019.2947146
32. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, United States (1994)

33. Genet., T., Jensen., T., Sauvage., J.: Termination of ethereum's smart contracts. In: Proceedings of the 17th International Joint Conference on e-Business and Telecommunications - SECRYPT. pp. 39–51. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009564100390051
34. Giesl, J., Schneider-Kamp, P., Thiemann, R.: Aprove 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) Automated Reasoning. pp. 281–286. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
35. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. Proc. ACM Program. Lang. **2**(OOPSLA) (oct 2018). https://doi.org/10.1145/3276486, https://doi.org/10.1145/3276486
36. Hu, T., Li, B., Pan, Z., Qian, C.: Detect defects of solidity smart contract based on the knowledge graph. IEEE Transactions on Reliability **73**(1), 186–202 (2024). https://doi.org/10.1109/TR.2023.3233999
37. Huang, Y., Wu, X., Wang, Q., Qian, Z., Chen, X., Tang, M., Zheng, Z.: The sword of damocles: Upgradeable smart contract in ethereum. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension. p. 333–345. ICPC '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3643916.3644426, https://doi.org/10.1145/3643916.3644426
38. Initiative, E.C.: Europe's Data Act: Implications for the Future of Innovation in Europe (2024), available: https://eu.ci/the-data-act-implications-for-the-future-of-smart-contracts-in-europe-annex/ (Accessed 08/2024)
39. Kwesili, O.: The Incompatibility of Self-Destruct Mechanisms in Upgradeable Smart Contract Architectures (2025), https://coinsbench.com/the-incompatibility-of-self-destruct-mechanisms-in-upgradeable-smart-contract-architectures-979f846f9265 (Accessed 07/2025)
40. Le, T.C., Qin, S., Chin, W.N.: Termination and non-termination specification inference. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 489–498. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737993, https://doi.org/10.1145/2737924.2737993
41. Le, T.C., Xu, L., Chen, L., Shi, W.: Proving conditional termination for smart contracts. In: Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts. p. 57–59. BCC '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3205230.3205239, https://doi.org/10.1145/3205230.3205239
42. Li, X., Yang, J., Chen, J., Tang, Y., Gao, X.: Characterizing ethereum upgradable smart contracts and their security implications. In: Proceedings of the ACM on Web Conference 2024. p. 1847–1858. WWW '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3589334.3645640, https://doi.org/10.1145/3589334.3645640
43. Marino, B., Juels, A.: Setting standards for altering and undoing smart contracts. In: Alferes, J.J., Bertossi, L., Governatori, G., Fodor, P., Roman, D. (eds.) Rule Technologies. Research, Tools, and Applications. pp. 151–166. Springer International Publishing, Cham (2016)

44. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers. p. 446–465. Springer-Verlag, Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-030-32101-7_27, https://doi.org/10.1007/978-3-030-32101-7_27
45. Mudge, N.: ERC-2535: Diamonds, multi-facet proxy (2020), ethereum Improvement Proposals, no. 2535, February 2020 [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-2535 (Accessed 08/2024)
46. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008), https://bitcoin.org/bitcoin.pdf Accessed: 06/2023
47. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. p. 653–663. ACSAC '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3274694.3274743, https://doi.org/10.1145/3274694.3274743
48. Olivieri, L., Mukherjee, A., Chaki, N., Cortesi, A.: Cross-chain Smart Contracts and dApps Verification by Static Analysis: Limits and Challenges. vol. 3962 (2025), https://ceur-ws.org/Vol-3962/paper16.pdf
49. Olivieri, L., Pasetto, L.: Towards Compliance of Smart Contracts with the European Union Data Act. In: CEUR Workshop Proceedings. vol. 3629, p. 61 – 66 (2024), https://ceur-ws.org/Vol-3629
50. Olivieri, L., Pasetto, L., Negrini, L., Ferrara, P.: European Union Data Act and Blockchain Technology: Challenges and New Directions. In: CEUR Workshop Proceedings. vol. 3791. CEUR-WS (2024), https://ceur-ws.org/Vol-3791/paper30.pdf
51. Olivieri, L., Spoto, F.: Software verification challenges in the blockchain ecosystem. International Journal on Software Tools for Technology Transfer (2024). https://doi.org/10.1007/s10009-024-00758-x, https://doi.org/10.1007/s10009-024-00758-x, published 2024/07/12
52. Olivieri, L., Spoto, F., Tagliaferro, F.: On-chain smart contract verification over tendermint. In: Bernhard, M., Bracciali, A., Gudgeon, L., Haines, T., Klages-Mundt, A., Matsuo, S., Perez, D., Sala, M., Werner, S. (eds.) Financial Cryptography and Data Security. FC 2021 International Workshops. pp. 333–347. Springer Berlin Heidelberg, Berlin, Heidelberg (2021)
53. OpenZeppelin: Erc-20 pausable (2024), available: https://docs.openzeppelin.com/contracts/4.x/api/token/erc20\#ERC20Pausable (Accessed 08/2024)
54. OpenZeppelin: Erc-20 snapshot (2024), available: https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#ERC20Snapshot (Accessed 08/2024)
55. OpenZeppelin: Erc-721 pausable (2024), available: https://docs.openzeppelin.com/contracts/4.x/api/token/erc721\#ERC721Pausable (Accessed 08/2024)
56. Patrick, C.: Principles of Abstract Interpretation. MIT Press Academic, Cambridge, MA, USA (2021)
57. Popper, N.: A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency. The New York Times (2016), june 17th
58. Rajasekar, V., Sondhi, S., Saad, S., Mohammed, S.: Emerging design patterns for blockchain applications. In: ICSOFT. pp. 242–249 (2020)
59. Ressi, D., Spanò, A., Benetollo, L., Piazza, C., Bugliesi, M., Rossi, S.: Vulnerability detection in ethereum smart contracts via machine learning: A qualitative analysis. arXiv preprint arXiv:2407.18639 (2024)

60. Rice, H.G.: Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathematical Society **74**, 358–366 (1953). https://doi.org/10.1090/s0002-9947-1953-0053041-6
61. Seneviratne, O.: The feasibility of a smart contract "kill switch". In: 2024 6th International Conference on Blockchain Computing and Applications (BCCA). pp. 473–480 (2024). https://doi.org/10.1109/BCCA62388.2024.10844477
62. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis. pp. 302–318. Springer International Publishing, Cham (2014)
63. Wasiu, A.: Mastering Solidity: require and Custom Errors in Ethereum Contracts (2023), https://medium.com/coinmonks/mastering-solidity-require-and-custom-errors-in-ethereum-contracts-b491565f1592 (Accessed 05/2025)
64. 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 (2018). https://doi.org/10.1109/IWBOSE.2018.8327565
65. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)