SmartML: Enhancing Security and Reliability in Smart Contract Development

Adele Veschetti[1], Richard Bubel[1] and Reiner Hähnle[1]

[1]Department of Computer Science, TU Darmstadt, Germany

Abstract

Smart contracts, as integral components of blockchain technology, play a pivotal role in automating and executing agreements in a trust free manner. However, ensuring the correctness and reliability of smart contracts remains a significant challenge due to their complex nature and potential vulnerabilities. To address this challenge, we propose SmartML, a novel modeling language tailored specifically to smart contracts. This paper presents an in-depth exploration of the features, advantages, and applications of the proposed modeling language. By providing a precise and intuitive means of describing smart contract behaviors, SmartML aims to support the development, verification, and validation of smart contracts, ultimately bolstering trust and confidence in them. Furthermore, we discuss the relation to our modeling language with existing blockchain technologies and highlight its potential to streamline integration efforts. We posit SmartML as a versatile tool to advance, interoperability, reliability, and security of smart contracts in blockchain ecosystems.

Keywords

smart contract, modeling language, reentrancy

1. Introduction

The primary objective of distributed ledgers, commonly implemented using blockchain technology, is to establish trust among various parties without relying on a central authority. Smart contracts play a pivotal role by encoding real-world transactions and their associated protocols, such as those outlined in legal agreements. These contracts are stored on the blockchain alongside their current state and are executed by individual nodes. For instance, a typical smart contract might manage an auction process, ensuring that the highest bidder rightfully secures the item while the auctioneer receives the funds once all conditions are met.

Smart contracts serve as the cornerstone of transactional integrity on a blockchain, operating under the principle of *code is law*. This underscores the critical question of whether smart contracts accurately fulfill their intended function. For instance, are auction rules correctly implemented, or could there be unintentional or malicious code paths enabling parties or external entities to bypass these rules and claim items or funds without fulfilling their obligations? The assurance of smart contract correctness is vital for achieving trust in blockchain infrastructures, as blockchains and consensus protocols alone do not address this crucial aspect. Numerous security vulnerabilities and high-profile attacks, such as the DAO attack that resulted in 50 million USD worth of Ether being lost, underscore the importance of this issue. In summary, ensuring trust in the accuracy of smart contracts is indispensable for the continued acceptance and responsible use of distributed ledger technology in society. Given the difficulty and expense of rectifying erroneous transactions post-execution due to blockchain's immutable nature, it is imperative to ensure the correctness of smart contracts before deploying them.

To address this challenge, we present a methodology designed to ensure the functional accuracy of smart contracts and identify potential bugs before their deployment. Our approach introduces the domain-specific, executable modeling language SmartML. It is designed to express the semantics of smart contracts and permits precise description of their intended behaviors. Such a *model-centric*

DLT 2024: 6th Distributed Ledger Technologies Workshop, May, 14-15 2024 – Turin, Italy

adele.veschetti@tu-darmstadt.de (A. Veschetti); bubel@cs.tu-darmstadt.de (R. Bubel); haehnle@cs.tu-darmstadt.de (R. Hähnle)

0000-0002-0403-1889 (A. Veschetti); 0000-0001-8000-7613 (R. Hähnle)

strategy offers several key benefits, such as comprehensibility, interoperability, and a lowered effort of formal verification.

In this paper, we outline the design of SmartML. We provide some background in Section 2, then justify and explain our approach in Section 3. The syntax of our modeling language is reported in Section 4, along with an example in Section 5. A representative example of our approach to reentrancy mitigation can be found in Section 6. An overview of similar approaches is in Section 7, followed by conclusions and next steps in Section 8.

2. Background

Blockchain technology facilitates a distributed computing architecture, wherein transactions are publicly disclosed and participants reach consensus on a singular transaction history, commonly referred to as a ledger [1]. Transactions are organized into blocks, timestamped, and made public. The cryptographic hash of each block includes the hash of the preceding block, creating an immutable chain that makes altering published blocks highly challenging.

Among the applications of blockchains, smart contracts stand out. These automated, self-executing contracts redefine traditional agreements, offering efficiency and transparency in various industries. A smart contract is a computer program delineated by source code written in a domain-specific language. It has the capability to automatically execute the terms of a distinct agreement expressed in natural language if certain conditions are met. Typically crafted in high-level languages, smart contracts are compiled to bytecode and encapsulated in self-contained entities deployable on any node within the blockchain. Smart contracts can be developed and deployed on various blockchain platforms, such as NXT [2], Ethereum [3], and Hyperledger Fabric [4]. Each platform offers distinct features designed for smart contract development, including specialized programming languages, contract code execution, and varying security measures.

Smart contracts, are susceptible to a range of vulnerabilities that can compromise their integrity and functionality. These vulnerabilities include reentrancy attacks, where a contract's code is re-entered before a previous function call completes, potentially allowing an attacker to manipulate the contract's state unexpectedly. Furthermore, improper access control mechanisms can lead to unauthorized access to contract functions or data, posing serious security risks. Given the immutable nature of smart contracts once deployed on a blockchain, it is paramount that they are rigorously tested and audited for correctness. Another significant issue of smart contracts is the handling of digital assets and tokens. Since blockchain transactions are irreversible and immutable, mismanagement or unauthorized access to these assets can lead to permanent loss. This necessitates robust mechanisms for securely managing and transferring assets while preventing duplication, leakage, or theft.

3. Our Approach

We propose a methodology that fosters comprehensibility of smart contracts, as well as their functional correctness and early detection of bugs *before* deployment. Our approach rests on a domain-specific, executable modeling language called SmartML, tailored to smart contracts. It permits precise description of their intended behavior. This model-centric approach offers important advantages over a traditional, implementation-centric one: (i) SmartML is intuitive, with a shallow learning curve, (ii) it has a formal and unambiguous semantics to prevent misunderstandings, and (iii) it is designed to facilitate static verification. By generating correctness certificates based on rigorous proofs, our methodology enhances trust in the accuracy of smart contracts. Moreover, our approach is fully interoperable with existing blockchain technology. Legacy smart contract languages, such as Solidity, can be translated from and to SmartML. In consequence, it is possible to develop, review, and verify smart contracts in SmartML, but deploy and execute them in a legacy environment. Our model-centric approach is visualized in Figure 1.

Existing (legacy) smart contracts can be represented and validated as SmartML contracts using automatic translation.

As SmartML contracts are based on a rigorous formal semantics, they have a clear and unambiguous semantics and are amenable for a wide range of static analysis. Security and functional properties can then be specified and verified producing independently verifiable certificates.

The verified models can then be compiled back into real-world smart contract languages like EVM bytecode, Solidity, etc. by provably correct compilation, which preserves the proven properties.

We explain the components of our approach now in more details: the design of SmartML is based on a careful analysis of the features and capabilities of a wide range of common smart contract languages, thereby identifying the most important ones to include. Overall, our methodology focused on synthesizing key elements of existing smart contract languages into a cohesive and expressive, fully executable modeling language, with a formal semantics and an expressive type system at its core.

In greater detail, SmartML meets the following requirements:

1. Permit clear and *concise* modeling of typical smart contract functionality to foster validation by code inspection.
2. Ensure *executability* so that behavior can be simulated and SmartML models can be compiled into native code such as the *Ethereum Virtual Machine*.
3. Provide enough *expressiveness*, so that legacy smart contracts can be translated into SmartML, specifically, to support (nested) programmable transactions.

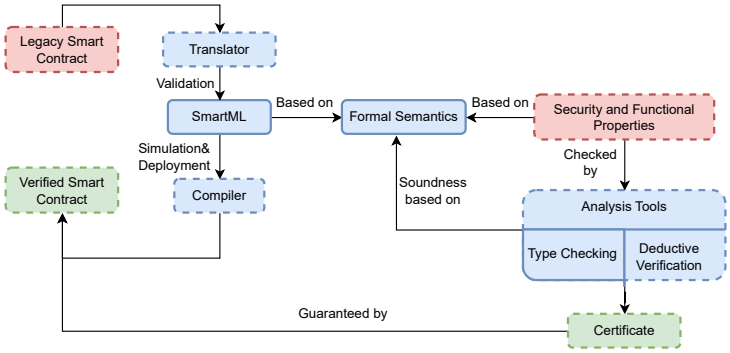

Figure 1: The SmartML Framework

After defining the core features, we proceeded to establish the semantics of the modeling language. This involved specifying the precise meaning and behavior of language constructs, ensuring clarity and consistency in their interpretation.

Additionally, we designed an expressive type system. By defining clear and consistent data types and their interactions within the language, we aimed to enhance the expressiveness and reliability of smart contract development. The purpose of the type system is to safeguard against reentrant calls through the implementation of locking mechanisms. When a function is invoked from either another contract or within the same contract, and it alters the contract's fields, it is automatically locked. This locking mechanism ensures exclusive access to the function, thereby mitigating reentrancy vulnerabilities.

So far we completed the initial design of SmartML, provided a formal semantics, and the type system [5]. In the future, our ecosystem will feature a translator capable of converting legacy smart contracts into SmartML models and back. This translator ensures interoperability of legacy contracts with SmartML, but also among different legacy formats.

Moreover, we will implement a deductive verification framework and a static analysis tool to ensure the security and functional correctness of smart contracts. This toolset will be designed to

- prove the absence of relevant classes of security vulnerabilities, including but not limited to reentrancy attacks, integer overflows, and unauthorized access;
- verify functional correctness of smart contracts by analyzing their behavior against formally specified requirements and business logic.

The deductive verification tool will be realized as an extension of the KeY system [6], the leading formal verification tool of the JAVA language, in whose design and development two of the authors are involved.

Our approach includes a reliable solution for securely storing and retrieving correctness certificates generated by our analysis tool. These certificates act as tangible evidence of adherence to both security standards and functional requirements. Stakeholders have the opportunity to access and authenticate these certificates, thereby ensuring the reliability and integrity of the smart contracts in question.

4. Syntax

The syntax grammar of SMARTML is in Table 1. We use a number of syntactic conventions for the symbols: C, D refer to contract names; A, L, K, M represent ADT declarations, contract declarations, constructor and method declarations, respectively; n stands for ADT function declarations; d indicates ADT expressions; f, g denote fields; m stands for method declarations; s, e, v, and τ cover statements, expressions, values, and types, respectively, while local variables are denoted by x. We use $\overline{f}$ to represent a sequence of elements $f_1, \ldots, f_n$.

The set of all local program variables is called ProgVars. For ease of presentation we assume that each local program has a unique name. The set of program variables ProgVars includes the special variable $\texttt{this}_C$ for each contract type C. If the context is clear the subscript C is omitted.

A SMARTML program consists of a series of ADT and contract declarations. The content of an ADT definition is formed by a sequence of function definitions. Permitted ADT expressions include the `if-then-else`, `return`, `switch`-constructs, and function calls.

P	$::= \overline{A}\ \overline{L}$	(program)
A	$::= \texttt{datatype}\ \ adt\ \{\texttt{constructor}\{\overline{fn} :: \overline{adt}\}\ \overline{F}\}$	(datatype)
F	$::= \tau_n\ n(\overline{\tau}\ \overline{w})\ \{d\}$	(ADT function)
d	$::= \texttt{if}\ (e)\ \{\ e\ \}\ \texttt{else}\ \{\ e\ \} \mid \texttt{return}\ e \mid n(\overline{w})$	(ADT expression)
	$\mid \texttt{switch}\ e\ \{\texttt{case}\ e :\ d;\ \texttt{default} :\ d\}$	
L	$::= \texttt{contract}\ C\ \texttt{extends}\ D\ \{\overline{f} : \overline{\tau}_f; K; \overline{M}\}$	(contract)
K	$::= \texttt{constructor}(\overline{g} : \overline{\tau}_g, \overline{f} : \overline{\tau}_f)\ \{\texttt{super}(\overline{g});\ \texttt{this}.\overline{f} = \overline{f}\}$	(constructor)
M	$::= \tau_m\ m(\overline{\tau}\ \overline{v})\ \{s\}$	(method)
s	$::= \texttt{if}\ (e)\ s\ \texttt{else}\ s \mid \texttt{while}(e)\ \{s\} \mid \texttt{let}\ e\ \texttt{in}\ s \mid s_1; s_2$	(statement)
	$\mid \texttt{try}\ (s)\ \texttt{catch}(\overline{x})\ \{s\} \mid v := e \mid v.f := e \mid v.m(\overline{v})$	
	$\mid \texttt{assert}(e) \mid \texttt{return}\ e \mid \texttt{try}\ s_0\ \texttt{abort}\ s_1\ \texttt{success}\ s_2$	
e	$::= v \mid !v \mid v.f \mid e_1\ \texttt{op}\ e_2 \mid e_1\ \texttt{bop}\ e_2$	(expression)
v	$::= x \mid \texttt{true} \mid \texttt{false} \mid \texttt{new}\ C(\overline{v}) \mid d$	(value)
op	$::= + \mid - \mid \times \mid \div$	(arithmetic operator)
bop	$::= \leq \mid \geq \mid\ \&\& \mid \parallel \mid = \mid \neq$	(boolean operator)

Table 1
The syntax of SMARTML

A contract declaration introduces a contract C that extends the contract D. The contract has fields $\overline{f}$ with the corresponding types $\overline{\tau}_f$, a constructor K and methods $\overline{m}$. The set of all contract types (names) is called Contract, the set of all fields is called Field. The constructor declaration is used to set up the initial values for the fields of a contract C. Its structure is determined by the instance variable declarations of C and the contract it extends: the parameters must match the declared instance variables, and its body must include a call to the super class constructor for initializing its fields with parameters $\overline{g}$. Subsequently, an assignment of the parameters $\overline{f}$ to the new fields with the same names as declared in C is performed. A method declaration introduces a method named m with return type τ_m and parameters $\overline{v}$ having types $\overline{\tau}$. Most statements are standard; for instance, $v := e$ and $v.f := e$ denote assignments, and $\texttt{assert}(e)$ checks certain conditions. If the expression e evaluates to `true`, executing an assertion is like executing a skip. On the other hand, if the expression evaluates to `false`, it is equivalent to throwing an error. The expressions are considered standard; however, for field access $v.f$, we restrict v to be only `this`. For ease of presentation, we assume that method invocations pass only local variables as arguments.

Compared to Solidity [7], SmartML provides a higher abstraction from the underlying execution engine like the Ethereum Virtual Machine (EVM) [8]. For example, no function calls via hashcode signatures are supported by SmartML. The semantics of SmartML is formally defined using structural operational semantics (SOS) [9]. Changes or new features need to update and extend the SOS rules. The advantage are that language features have an unambiguous behaviour and remain amenable to static analysis. By supporting algebraic datatypes to realise common datastructures like list and maps, we avoid certain shortcomings that are present in Solidity (no iteration over key-value pairs; only cryptographic guarantees that content put in a map does not overwrite other data) due to their need to work with a concrete blockchain. Explicit transaction primitives (try–success–abort) allow us to have a syntactic marker for function calls opening a new transaction as well as direct error handling.

5. Example

We discuss the syntax and rationale behind the design decisions made for SmartML[1] along the implementation of a simple smart contract (shown in Listing 1) modelling a store.

Lines 1–17 define a list of integers as an algebraic data type (ADT). The actual store is defined as contract Store (lines 18–44) with two fields: the list of customer addresses and their corresponding balances managed as a list of integers.

Algebraic data types are mathematically well-understood. We use them to describe data structures and to implement their operations in a functional style. The fact that ADTs are immutable and their operations are pure (i.e., side-effect free), provides advantages for the developer of a SmartML contract. It also makes static analysis, in particular, deductive verification, easier and improves the degree of automation. By allowing ADT definitions, we let users create custom data structures tailored to their modeling needs, improving the expressiveness and flexibility.

We focus on two functions of the contract: deposit, which lets customers deposit funds in their balances, and withdraw that allows them to withdraw deposited resources. The following concepts are needed to understand the presented model:

1. Each contract function has an implicit parameter sender to access its caller.
2. SmartML supports (for the moment) one user-defined resource. If none is defined, a predefined resource type, called Coin, is available.
3. Contract functions can have a resource parameter $[c : Resource\mathsf{Info}]$ to receive the specified resource. A $Resource\mathsf{Info}$ represents a data container specifying the resource type and the amount sent. The amount is deposited to the callee before execution of the called function starts.
4. Contracts can access the amount of a resource they possess using the implicit field balance, which can be viewed as a read-only field (it updates upon transferring resources via function calls).

The deposit function declares the parameter r, signaling that it anticipates explicit resource sending. An illustration of an external call with explicit resource consumption can be found in line 38. We point out that each contract includes a predefined receive method that cannot be overwritten. We do not report the definition of ListAddress, as it closely follows the structure outlined for ListInt , as previously mentioned. Furthermore, we have omitted the details of standard setter and getter methods.

The functions deposit and withdraw receive and send resources via transactions, respectively. Lines 38–43 demonstrate how transactions are explicitly handled using a try–abort–success statement. Supporting explicit transaction initiation and finalization, particularly for nested transactions, in SmartML offers developers granular control over transaction execution. This approach helps error handling, resource management, and overall robustness and reliability of the code.

Using deductive verification, we can express complex properties beyond those guaranteed by type checking or model checking approaches. For instance, in our example (assuming that the ADT provides a length and getter function) we can write, for example,

$$\text{this.balance} = \mathsf{sum}\{\text{int i;}\}(0,\ \text{i<length(this.balances)},\ \text{this.balances.get(i)})$$

[1] early version available at https://projects.se.informatik.tu-darmstadt.de/projects/gitlab/athene-smartcontract/smart-ml

to express that the balance of the Store contract is equal to the sum of the coins stored by its clients. This is a typical invariant that must hold at certain points which is provable by deductive verification.

```
datatype ListInt {
  constructor{ nil :: ListInt | cons(int v, ListInt tail) :: ListInt}
  int indexOf(ListInt l, int n){
    switch (l) {
    case nil: return -1;
    default: if (l.v == n) { return 0; }
             else {
                int indexInTail = indexOf(l.tail, n);
                switch (indexInTail) {
                  case -1: return -1;
                  default: return indexInTail + 1;
                }
             }
    }
  }
  ListInt addElement(ListInt list, int element) { return cons(element, list); }
}
contract Store {
  ListAddress addresses;
  ListInt balances;
  constructor(){
    this.addresses = nil;
    this.balances = nil;
  }
  function deposit([r : CoinInfo]) {
    int index = addresses.indexOf(addresses,sender);
    if (index != -1) { balances.set(index,balances.get(index) + r.amount); }
    else {
      addresses = addresses.addElement(addresses,sender);
      balances = balances.addElement(balance,r.amount);
    }
  }
  function withdraw() returns bool {
    int bal = 0;
    int index = addresses.indexOf(addresses,sender);
    if (index != -1) { bal = balances.get(index); }
    assert(bal > 0);
    try sender$bal.receive();
    abort { return false; }
    success {
      balances.set(index,0);
      return true;
    }
  }}
```

Listing 1: SmartML code example

While SmartML may appear at first glance to be slightly verbose, this verbosity is a deliberate design choice aimed at ensuring clarity, precision, and unambiguous interpretation of the code. By providing a formal semantics, we offer developers a rigorous framework for understanding the exact meaning and behavior of each construct within the language. The formal semantics serves as a foundation for reasoning about the correctness and properties of programs written in the language, offering assurance that code behaves as intended under all possible circumstances. Moreover, by providing clear and precise rules for program behaviour and data manipulation, SmartML's formal semantics enhance program clarity and reduce the likelihood of errors. For instance, when defining functions or data structures, programmers can rely on the formal semantics to ensure consistent and predictable behavior.

SmartML is more explicit compared to more concise alternatives, but this approach ultimately fosters confidence in the reliability and predictability of the resulting model. SmartML is a modeling language and not a programming language: Once trust in the model is established, compiling a SmartML model into a concrete smart contract language optimized for efficiency, can perform optimizations as part of a provably correct compilation process.

6. Reentrancy Mitigation

In this section, we provide a brief overview of the type system. While we do not detail the rules, we offer an illustrative example demonstrating how it effectively prevents reentrancy.

To guarantee the secure execution of a SmartML program, we have implemented a type system aimed at mitigating unsafe reentrancy while allowing for provably safe reentrant calls. Our goal with this type system is to prevent reentrant calls through the implementation of robust locking mechanisms. Whenever a function is invoked from either another contract or from within the same contract, which in turn modifies the contract's fields, it is automatically flagged as 'locked'. This ensures exclusive access and prevents reentrancy vulnerabilities.

Contract identities and locked methods are monitored to effectively handle aliasing, ensuring that each reference is correctly managed within the program's memory. This tracking process, essential for maintaining data integrity, involves closely observing the interactions between different parts of the program to prevent unintended side effects.

We now present an simple example of single function reentrancy, where a function within the contract can be called recursively before the completion of its initial execution. Additionally, in [5], we show how the type system effectively prevents not only single-function reentrancy but also cross-function and cross-contract reentrancy.

Figure 2: Call graph for a cross-function reentrancy example

In the example presented in Figure 2, the interaction between the Store and Attacker are:

1. Initial Trigger: The `attack` function of the Attacker contract is invoked, initiating the interaction. This triggers a call to the `withdraw` function of the Store contract.
2. Interaction with Store Contract: Upon receiving the call from the Attacker contract, the `withdraw` function of the Store contract is executed. Inside the `withdraw` function, a call to the `receive` function of the Attacker contract is made.
3. Reentrancy Exploitation: Within the `receive` function of the Attacker contract, another call to the `withdraw` function of the Store contract is initiated. This recursive call perpetuates the reentrancy vulnerability, potentially leading to further exploitation of the vulnerability.

Our type system effectively prevents this repetitive call by leveraging the set Δ. Specifically, during the type system's verification process to determine whether the function `withdraw` is permissible, the derivation encounters a failure. This failure occurs because the objects representing the current instance of the `Store` contract and the external contract being interacted with are identified as being aliased within the system. This identification serves as a crucial indicator that a reentrant call is in progress or has the potential to occur, prompting the type system to block the operation and safeguard against unauthorized or unintended actions within the smart contract.

While a complete ban on reentrancy seems like a simple solution, it is overly restrictive and will drastically reduce the functionality and interoperability of smart contracts, thus limiting their potential. For this reason, our type system has been carefully designed as a safeguard against unsafe reentrant calls while permitting those considered to be secure. A key feature is the ability to assess whether a call

to an external contract occurs after all necessary checks and updates to the fields have been executed. When such a call satisfies the non-interference condition, it is considered safe. Such calls are not added to the set Δ of locked method calls. Thus, this approach to designing our type system serves a dual purpose: it effectively prevents reentrancy attacks while enabling the execution of safe calls, finding a middle ground that avoids unnecessary restrictions.

7. Related Work

Several approaches to verification and security analysis of smart contracts were suggested, ranging from bug finding to deductive verification. Middleweight static analyzers employ advanced techniques such as symbolic execution, control-flow analysis, and taint analysis to pinpoint common security vulnerabilities in smart contracts, including issues like unsafe reentry and mutable control flow. Examples of approaches and tools in this category include Oyente [10], Securify [11], Maian [12] and EthBMC [13]. They focus on discovering bugs but may overlook certain program paths, while our approach is intended to guarantee security against cross-contract reentrancy attacks. In particular, the majority of those static analyzers function as bug-finding tools, aimed at reducing the number of contracts falsely identified as defective (false positives). These tools typically employ symbolic execution to achieve their objective. Additionally, they strive to prove the security of smart contracts, aiming to minimize false negatives while providing verifiable assurances for their analysis results. Notably, eThor [14], as far as we are aware, is the only tool known to offer a provable soundness claim, utilizing abstract interpretation for its analysis. However, numerous security properties for smart contracts necessitate comparing execution traces from distinct initial configurations, falling under the broader category of 2-safety properties. Checking 2-safety properties with tools restricted to analyzing reachability properties, like eThor, requires an overestimation of the original property in terms of reachability.

A deductive approach is taken in [15]. It uses a specification language based on temporal logic. In [16], the authors perform model checking using SPIN and translate smart contracts into ProMeLa programs. A similar approach is taken in [17], where smart contracts, the underlying blockchain and miners are translated into timed automata. Formal verification tools like VERISOL [18] and Solythesis [19] generate proofs and identify counterexamples, ensuring smart contracts adhere to access control policies and invariant specifications, respectively. However, these tools focus on single contract safety and lack compositional verification capabilities. Further, our deductive verification approach permits to formalize and prove complex first-order properties that go beyond what can be shown by model checking.

8. Conclusion and Next Steps

SmartML is a language-independent modeling and verification framework for smart contracts. It offers a comprehensive approach to formally specifying and verifying smart contracts, addressing the inherent complexities and vulnerabilities that can pose risks to security-critical assets.

The platform-independent modeling language complements existing state-of-the-art approaches by providing a structured and abstract representation of contracts, facilitating understanding and analysis. To ensure full platform independence, we are currently developing a translator that converts existing smart contract languages to SmartML and back.

SmartML has a formal operational semantics and is equipped with a type system designed for safe reentrancy checks, establishing a foundation for expressing and verifying both functional correctness and security properties of smart contracts. Additionally, we plan to develop a deductive verification tool and a static analysis tool to prove the absence of relevant classes of security vulnerabilities and ensure functional correctness of smart contracts, which are part of our future work.

Acknowledgments

This work was funded by the ATHENE project "Model-centric Deductive Verification of Smart Contracts".

References

[1] I. Bashir, Mastering Blockchain, Packt Publishing, 2017.

[2] NXT, official website, https://nxt.org/.

[3] Ethereum, official website, https://ethereum.org/.

[4] Hyperledger Fabric, official website, https://www.hyperledger.org/projects/fabric.

[5] A. Veschetti, R. Bubel, R. Hähnle, Smartml: Towards a modeling language for smart contracts. arXiv:2403.06622.

[6] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich (Eds.), Deductive Software Verification—The KeY Book: From Theory to Practice, volume 10001 of *LNCS*, Springer, 2016.

[7] Solidity by Example - Reentrancy, https://solidity-by-example.org/hacks/re-entrancy/.

[8] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[9] G. D. Plotkin, A structural approach to operational semantics, J. Log. Algebraic Methods Program. 60-61 (2004) 17–139.

[10] L. Luu, D. Chu, H. Olickel, P. Saxena, A. Hobor, Making Smart Contracts Smarter, in: CCS, ACM, 2016, pp. 254–269.

[11] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, M. T. Vechev, Securify: Practical security analysis of smart contracts, CoRR abs/1806.01143 (2018).

[12] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, A. Hobor, Finding the greedy, prodigal, and suicidal contracts at scale, Proceedings of the 34th Annual Computer Security Applications Conference (2018). URL: https://api.semanticscholar.org/CorpusID:3613721.

[13] J. C. Frank, C. Aschermann, T. Holz, Ethbmc: A bounded model checker for smart contracts, in: USENIX Security Symposium, 2020. URL: https://api.semanticscholar.org/CorpusID:215805835.

[14] C. Schneidewind, I. Grishchenko, M. Scherer, M. Maffei, ethor: Practical and provably sound static analysis of ethereum smart contracts, CoRR abs/2005.06227 (2020).

[15] A. Permenev, D. K. Dimitrov, P. Tsankov, D. Drachsler-Cohen, M. T. Vechev, Verx: Safety verification of smart contracts, in: SP, IEEE, 2020, pp. 1661–1677.

[16] X. Bai, Z. Cheng, Z. Duan, K. Hu, Formal modeling and verification of smart contracts, in: ICSCA, ACM, 2018, pp. 322–326.

[17] T. Abdellatif, K. Brousmiche, Formal verification of smart contracts based on users and blockchain behaviors models, in: NTMS, IEEE, 2018, pp. 1–5.

[18] S. K. Lahiri, S. Chen, Y. Wang, I. Dillig, Formal Specification and Verification of Smart Contracts for Azure Blockchain, CoRR abs/1812.08829 (2018).

[19] A. Li, J. A. Choi, F. Long, Securing smart contract with runtime validation, in: PLDI, ACM, 2020, pp. 438–453.