Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control

Jonas Schiffl
Karlsruhe Institute of Technology
Institute of Information Security and Dependability (KASTEL)
jonas.schiffl@kit.edu

Matthias Grundmann
Karlsruhe Institute of Technology
Institute of Information Security and Dependability (KASTEL)
matthias.grundmann@kit.edu

Marc Leinweber
Karlsruhe Institute of Technology
Institute of Information Security and Dependability (KASTEL)
marc.leinweber@kit.edu

Oliver Stengele
Karlsruhe Institute of Technology
Institute of Information Security and Dependability (KASTEL)
oliver.stengele@kit.edu

Sebastian Friebe
Karlsruhe Institute of Technology
Institute of Telematics
sebastian.friebe@kit.edu

Bernhard Beckert
Karlsruhe Institute of Technology
Institute of Information Security and Dependability (KASTEL)
beckert@kit.edu

ABSTRACT

Ethereum is a platform for deploying smart contracts, which due to their public nature and the financial value of the assets they manage are attractive targets for attacks. With asset management as a main task of smart contracts, access control aspects are naturally part of the application itself, but also of the functions implemented in a smart contract. Therefore, it is desirable to establish the correctness of smart contracts and their access control on application and single-function level through formal methods. However, there is no established methodology of formalising and verifying correctness properties of smart contracts. In this work, we make an attempt in this direction on the basis of a case study. We choose an existing smart contract application which aims to ascertain the integrity of binary files distributed over the Internet by means of decentralised identity management and access control. We formally specify and verify correctness at the level of single functions as well as temporal properties of the overall application. We demonstrate how to use verified low-level correctness properties for showing correctness at the higher level. In addition, we report on our experience with existing verification tools.

ACM Reference Format:
Jonas Schiffl, Matthias Grundmann, Marc Leinweber, Oliver Stengele, Sebastian Friebe, and Bernhard Beckert. 2021. Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies (SACMAT '21), June 16–18, 2021, Virtual Event, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3450569.3463574

1 INTRODUCTION

Deployed smart contracts are programs that run on decentralised platforms on which they manage resources and provide services. The most prominent smart contract platform at present is Ethereum. Ethereum smart contracts are usually written in the Solidity programming language. They manage Ether, Ethereum's built-in cryptocurrency, as well as other digital and a wide range of real-world assets. Thus, smart contracts are per se access control applications since they manage access to assets. This fact makes them a target of attackers seeking financial gain [10]. Since every smart contract's byte code is publicly available on the Ethereum blockchain, any vulnerability – either on the level of the entire application or on underlying functions of a smart contract – is likely to be found by some attacker. Furthermore, once deployed, Ethereum smart contracts are immutable; the only way to fix programming errors is to delete, correct, and redeploy the entire program.

As a consequence, it is necessary that smart contracts are correct upon deployment, i.e., they work as intended by the developer and cannot be exploited by an attacker. This requirement of correctness is particularly important for security properties and corresponding access control aspects. In order to ensure correctness, formal methods should be used: correctness of a contract is expressed in the form of formal specification and is proven to show that the implementation indeed fulfils the specification. The use of formal verification comprises addressing the following questions for each smart contract application: What are the relevant functional and security properties? How can those properties be formally specified? How can one prove that the application conforms to its specification? A long-term goal and a "grand challenge" is to develop a methodology for translating an intuitive phrasing of what the application is intended to do into a formal language, and to then prove that the implementation conforms to the formalisation. The challenge lies in the fact that applications based on a collection of smart contracts are 'service-like programs' without a deterministic start-to-end program flow. Functions of a smart contract are called by unknown, untrusted agents in an unknown order, so that the overall system behaviour is best described as non-deterministic. Function execution, however, is deterministic and atomic.

In this work-in-progress paper, we make an attempt towards the ambitious goal of correct smart contracts, focusing on a specific application. As case study, we choose Palinodia [11, 12], an application aiming to enforce access control over the publication and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SACMAT '21, June 16–18, 2021, Virtual Event, Spain
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8365-3/21/06…$15.00
https://doi.org/10.1145/3450569.3463574

revocation of information concerning the integrity of executable binary files distributed over the Internet. Our contributions consist of an analysis what kinds of properties are relevant for smart contract correctness, a strategy for proving correctness on application-level based on correctness on function-level, and an evaluation of the applicability of available tools for formal analysis.

We proceed as follows: After reviewing related work on verification of smart contracts and fundamental concepts of the Palinodia application (Section 2 and Section 3), we start at the 'bottom' and specify as well as verify the correctness of single Solidity functions (Section 4). In particular, we develop a natural language specification of each of Palinodia's functions and translate them to a formal specification language. Then, we verify the correctness of the implementation using the SOLC-VERIFY tool. In Section 5, we formulate the application's main functional and security properties. These properties are not on the level of single functions, and cross the boundary between the smart contracts of the application and the client software: The crucial point of the application is whether a user is able to validate the integrity and the correct endorsement of a binary. We formalise this validation check by translating it into a Solidity smart contract. From there, we reason about the implications of a program state where the validation returns positive for correctness and security. Due to the non-deterministic agent behaviour, temporal verification methods are needed. We experiment with using the VERX tool to formalise some temporal correctness properties, but we have not yet been successful at using the tool to prove them correct. However, we illustrate how to use the correctness properties verified at the single-function level to manually prove correctness of the application's main security property, and give some thoughts on how to automate or formalise this process in the future. Finally, in addition to formalising functional and security properties of individual smart contracts, it can be useful to check for application-independent correctness properties, e.g., vulnerabilities or anti-patterns arising due to the idiosyncrasies of blockchain networks or the Solidity programming language. Various tools exist that aim to discover these kinds of vulnerabilities, usually through automated static analysis. In Section 6, we describe our experience with two tools of this category.

2 RELATED WORK

Formal verification of smart contracts has been an active research area. A recent overview [1] notes, however, that none of the surveyed tools support application-specific annotations, and that none of the tools is able to verify properties of interacting smart contracts. While some such tools do exist, this is still an indication that specification and verification of the overall correctness of smart contract applications should be investigated further.

We are aware of only a few tools which allow user-defined specification. The SOLC-VERIFY [2] tool allows specifying and verifying function pre- and postconditions and invariants. We describe it in more detail in Section 4.

The VERX tool [8] allows user-defined specification of temporal properties, written in PastLTL. The complexity of function calls is dealt with through predicate abstraction. We describe some initial experiences with VERX in Section 5.3.

The $\mathbb{K}$ framework is a general approach for generating analysis tools from an operational semantics of a language. KEVM [4], a formal semantics of Ethereum for the $\mathbb{K}$ framework, has been developed to allow formal verification of Ethereum smart contracts on the EVM byte code level. As a very general tool, KEVM in principle allows the specification of functional as well as safety and security properties. However, its generality comes with a trade-off in specification complexity and performance of the automated proof tools.

Most of the highly publicised exploits of smart contracts were not due to application-specific properties, but due to errors like reentrancy or missing parameter sanitization. Generic vulnerabilities of this kind have been compiled in the Smart Contract Weakness Classification Registry[1], which has become a benchmark suite for static analysis tools.

Formal verification of correctness ("safety") of access control specifications dates back at least to the seminal work of Harrison et al. in [3]. While the fundamental limits certainly apply to the grand challenge and this work alike, formal verification of selected systems or classes of systems can still be feasible. In this work, we are particularly interested in formalising the properties of typical smart contracts, and in checking whether existing tools are able to verify them.

3 PALINODIA

Palinodia [11, 12] is the application we chose for our case study. It can be used to establish unique identities for software and enforce access control over these identities, including the publication and revocation of integrity protecting information for individual binaries. Broadly speaking, it codifies the relationship between a Software Developer, who establishes and maintains a software project, and a Software Maintainer, who is authorised by the Developer to produce executable binaries from the software project. Palinodia is comprised of three kinds of smart contracts as depicted in Figure 1: A *Software* contract establishes a root identity for a software product and is controlled by a Software Developer via an Identity Management contract. It can store references to several distinct Binary Hash Storage (BHS) contracts (representing different intermediary identities of the software), each managed by a different Maintainer. A *Binary Hash Storage* contract represents an intermediary identity of a software product and is managed by a Maintainer. Hashes of binaries (stored elsewhere) can be published, representing an endorsement by the Maintainer. They can also be revoked later. Each BHS contract is associated to one Software contract and one Identity Management contract. *Identity Management* contracts are used by Software Developer and Maintainer alike to control who has access to the functions of the software and BHS contracts respectively. In particular, individual Ethereum public keys can be added to and removed from Identity Management contracts to authorise or deauthorise them.

User clients can obtain contracts and their current state from the Ethereum network in order to verify both the integrity and endorsement of binaries they wish to use. To facilitate this, binaries include a metadata manifest with the address of their respective BHS contract. After obtaining the current state of the BHS contract,

[1] https://swcregistry.io/

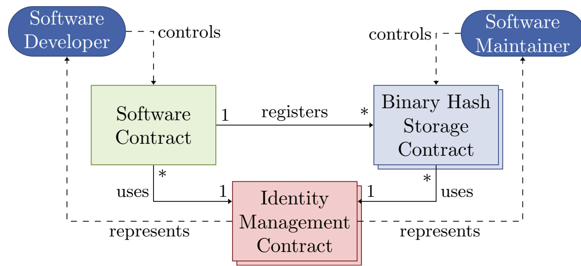

Figure 1: Overview of Palinodia, consisting of two roles (blue pills) and three kinds of smart contracts (coloured rectangles) and their mutual relations. Source: [11]

which includes the published hash of the binary for comparison, the user client proceeds to check whether the BHS contract is endorsed by the Software contract it expects. The first time a user obtains a binary, the address of its corresponding Software contract is stored as a trust anchor. Any binary that links back to such a stored Software contract through a BHS contract is trusted by the user client.

Simply put, the Palinodia user client trusts data it reads from the blockchain because it assumes that this data has been deposited in the intended way by authorised agents. The goal of our formal analysis is to formalise and prove correctness of this assumption.

4 SINGLE FUNCTION PROPERTIES

As a starting point for the formal analysis of the Palinodia application, a natural language specification for each function of the three smart contracts was developed in the form of pre- and postconditions. All preconditions are in one of two categories: (1) The caller must have access rights to the function, and (2) checking whether a storage variable is already set to a specific value. The postconditions specify that a storage variable must not be modified if the preconditions are not fulfilled.

This kind of specification is easily mapped to the `require` mechanism in Solidity. The `require` keyword takes a boolean condition. If the condition is fulfilled, execution continues; if it is not, execution is aborted and all changes to state are reverted.

4.1 Example Specification

The natural language specification of the `changeRootOwner` function was given in the following form:

> `Change Root Owner (Parameter: New Root Owner's public key)` *The current Root Owner can transfer this role to a different public key.*
> ***Precondition****: The caller is the current Root Owner.*
> ***Postcondition****: The given public key is the new Root Owner. The previous Root Owner loses their status.*
> ***Precondition****: The caller is not the current Root Owner.*
> ***Postcondition****: No state change.*

We translate the natural language specification to annotations for the SOLC-VERIFY tool. The function's source code, annotated with a postcondition describing its desired effect, is given in Listing 1.

```
//@ postcondition root_owner = _newAddress;
function changeRootOwner(address _newAddress)
    public returns(bool) {
  require(msg.sender == root_owner,
    "Operation_only_allowed_for_Root_Owner");
  root_owner = _newAddress;
  return true;
}
```

Listing 1: The `changeRootOwner` function

4.2 The SOLC-VERIFY Tool

Functions specified in this way can be proven correct with SOLC-VERIFY [2]. SOLC-VERIFY is a tool for formal verification of Ethereum smart contracts on the Solidity level. Its annotation language allows specifying contract-level invariants, function-level pre- and postconditions, loop invariants, and frame conditions.

The `require` and `assert` statements in the Solidity source are translated to pre- and postconditions, respectively. Annotations can contain quantifiers, and bounded sum terms. Function postconditions can refer to the return value of the function and to the *old* value of a state variable before the function was executed. `modifies` clauses in the specification of a function express which part of the state may be changed by the function.

The tool works by translating the source code and the annotations to the Boogie intermediate language, generating verification conditions which can be discharged by the z3 SMT solver. While specification is supplied by the developer, verification is automatic.

4.3 Verification using SOLC-VERIFY

After translating the natural language specification into SOLC-VERIFY's annotation language, we were able to verify the correctness of almost all functions w.r.t. the specification.

Only in a few cases did specification or verification not work as expected. One problem is String comparison: One function takes a String parameter and uses it to replace a mapping entry. The postcondition of the function would need to express equality between the parameter and the mapping entry after execution of the method. However, there is no method to directly compare Strings either in Solidity or in the annotation language of SOLC-VERIFY.

When specifying the frame of a function, i.e., what parts of the state a function may change, SOLC-VERIFY times out in some instances when attempting to prove that only one element of a data structure can be changed. For the proof to succeed, it is necessary to allow the entire data structure to be changed.

Apart from these issues, using SOLC-VERIFY is straightforward and fast. The specification needs as much space as the source code. Automatic verification of all 30 functions takes around 30 seconds.

4.4 Single Function Access Control

Much of the natural language specification for Palinodia concerns access control. Access control failures also account for many critical vulnerabilities in deployed smart contracts (cf. [6, 7]). For this reason, we highlight some general concepts for achieving correct access control.

Palinodia's built-in access control is role-based. There are three different *roles*: Root Owner, Software Maintainer, and Software

```
contract RequireCheck {
  function danger(uint256 i) public returns(bool) {
    require (msg.sender == owner, "Caller␣is␣not␣owner");
    doDangerousThings();
  }
}
```

Listing 2: Access control by requires clause

```
contract ModifierCheck {
  modifier only_owner {
    require (msg.sender == owner, "Caller␣is␣not␣owner");
    _;
  }
  function danger(uint256 i)
        public only_owner returns(bool) {
    doDangerousThings();
  }
}
```

Listing 3: Access control by modifier

Developer. Each of these roles has an associated set of functions which may be called by someone who has that role.

Checking whether the caller of a function has a certain role is currently done in a `requires` clause in the beginning of each function. For the Root Owner, the address of the caller is compared to the Root Owner address stored in the contract. For the other roles, calls to the Identity Management contract are made. The `requires` clauses could be replaced by *function modifiers*, a Solidity concept which is useful to replace recurring checks with a single keyword. Listing 2 and Listing 3 show how access control can be implemented via `requires` statement and function modifiers, respectively.

The `only_owner` modifier can be reused, reducing the amount of code, and is more readable. It also simplifies the task of verifying that access control happens correctly, because only 3 modifier implementations (and their presence in the function header) have to be checked (as opposed to a requires clause for every function).

In [9], there is an approach for automatically generating modifiers and function stubs from an existing access control model of the application which includes the roles and the functions which each role is allowed to access. Furthermore, the report describes how access to storage can be modelled and verified.

In conclusion, specifying and verifying the correctness of single functions has worked well in our case study. Palinodia does not contain any loops or recursion, which typically present a challenge for verification tools. In the future, we will need to investigate the performance of existing tools when dealing with these constructs as well.

5 APPLICATION CORRECTNESS

While Section 4 dealt with the correctness of single functions, in this section, we consider the correct and secure functioning of the application as a whole.

5.1 Intuitive Security Property

The security of the application relates the smart contracts and the validation a user performs in order to ensure that the piece of software they downloaded has not been tampered with. It describes what a user expects from the software and can be stated as follows:

Security*: If the client's validation check for a binary is positive, then the binary is endorsed by the Software Developer who has been chosen to be trusted on first use of this software.*

We call this the security property of the application. Note that we refer to the Software Developer as a person, which is a simplification. In Ethereum, persons are identified by addresses, which are derived from a person's public key that corresponds to their private key. In Palinodia, there is a specific contract for identity management, where each identity (e.g., a Software Developer or a Software Maintainer) is bound to a public key. For better readability, we abstract from this layer and we refer to these identities directly as persons in the following.

The functional correctness property is very similar to the security property, but in the other direction:

Functional Correctness*: If a binary is endorsed by a Software Developer who has been chosen to be trusted on first use of this software, the client's validation check for a binary is positive.*

In the following, we will focus on verification of the security property, as we consider it more critical property. We leave verification of the functional correctness property for our upcoming work.

5.2 Formalising the Client Validation

In order to reason about the main properties, we need a better grasp of the client-side validation. The client validation has the following five parameters: `local_sw` is the address of the Software contract where the user first downloaded the software. This value is 0 on first use when it is retrieved and stored. On every subsequent validation, the stored value is used as input. `local_hash` stores the hash of the downloaded binary that the user computed locally. `bhs` represents the address of the BHS contract (included in the metadata associated with the binary). `hash_id` is the ID where the hash of the binary is stored (included in the metadata associated with the binary). `software_id` represents the ID of the software (included in the metadata associated with the binary).

From the BHS contract, the client retrieves the address of the Software contract `remote_sw`. The client validation returns true if

1. the hash of the downloaded binary as locally computed by the client (`local_hash`) is published in the BHS contract (`bhs`), and
2. the Software contract (`remote_sw`) referenced by the BHS contract (`bhs`) links back to (endorses) the BHS contract (`bhs`), and
3. the Software contract (`remote_sw`) is the correct one, i.e., it matches the one stored locally by the client (`local_sw`).

In an attempt to capture the properties of the application state which are relevant to the client validation, we encoded the process of a client attempting to validate the integrity of a binary file as a Solidity smart contract. The contract's storage contains the names and addresses of Software contracts which the client trusts. The contract consists only of a single function `validate()`. The locally computed hash and the needed addresses are modelled as parameters to this function. Having specified the client validation in the

```
□(bhs_hash != 0
  => ◇(Function == BHS.publishHash(string, uint256) &&
    BHS.publishHash(string, uint256)[1] == bhs_hash)))
```

Listing 4: If `bhs_hash` (the hash value stored in the BHS contract) is non-zero, then someone must have called `publishHash()`.

```
□(bhs_hash == 0
  => (□(bhs_hash == 0)
    || ◇(Function == BHS.revokeHash(string) &&
     BHS.revokeHash(string)[0] == bhs_hash)))
```

Listing 5: If `bhs_hash` is zero, then either it was always zero, or someone must have called `revokeHash()`.

same language as the smart contract is a basis for using a verification tool that proves the security property because the security property references the client's validation check.

5.3 Temporal Formalisation

We now want to express a relationship between a state of the application in which the client validation is positive, and the history leading to this state. This suggests a temporal formalisation.

VerX [8] is a tool which operates on Solidity programs annotated with temporal formulas. The annotations are written in Solidity enriched with the *past temporal logic* (PLTL) operators □ (*always*), ◇ (*once*) and • (*previously*). In the formalisation of correctness properties in Listing 4 and Listing 5, we adopt this dialect. These properties give a meaning to whether or not a hash value in the BHS is 0: If it is not, then someone called the `publishHash()` function. If it is 0, then it has either always been 0, or the `revokeHash()` function was called with this value as a parameter.

The `publishHash()` and `revokeHash()` functions are access controlled: they can only be called by addresses who have maintainer status in the associated project. In order to formalise the entire security property, we would now need to express that an address which is authorised to make the function call is either the address which was trusted on the first download of the binary, or was (transitively) endorsed by this address.

VerX's specification formalism is expressive and intuitive. However, we are still experiencing practical issues with actually proving the security property of the application w.r.t. the temporal specification. We are in contact with the VerX developers and hope to resolve these issues.

Fortunately, we can provide a manual proof in the following subsection.

5.4 Pen-and-Paper Proof of the Security Property

Our goal is to automatically prove the application level security property by relying on low-level proofs on single function level. To get an impression of how such a proof is built and what assumptions and statements are required as input, we manually prove the security property of the Palinodia application in this section. We use the definition of the security property of the Palinodia application from above and show that it is true using temporal reasoning and properties verified at the single-function level. To introduce notation for the proof, we repeat the security property here:

***Security**: If the client's validation check for a binary b is positive, then the binary is endorsed by the Software Developer* `developer_sw` *who has been chosen to be trusted on first use of this software.*

To show this statement, we assume the first part (successful validation of a binary) and deduce the second part (binary is endorsed by a Software Developer). A binary is endorsed by a Software Developer if it has previously been endorsed and has not been revoked. According to Section 5.2, a positive verification result for a binary b associated with a BHS contract `bhs` implies these statements:

- (S1) The Software contract `remote_sw` associated with the given BHS contract `bhs` has been added to the list of trusted Software contracts.
- (S2) The Software contract `remote_sw` endorses the given BHS contract `bhs`.
- (S3) The hash $H(b)$ is registered with the BHS contract `bhs` and has not been revoked.

We further need the following assumptions about the application's context:

- (A1) A user who trusts a Software contract trusts the Root Owner and the Software Developer associated with this Software contract.
- (A2) A Software Developer who endorses a BHS contract trusts the Software Maintainer and Root Owner associated with this BHS contract (i.e., these entities are allowed to publish binaries for the software or grant this right to other entities).
- (A3) Endorsement / Trust is transitive: If a Software Developer endorses a Software Maintainer and a Software Maintainer endorses a hash, then the Software Developer endorses the hash.

From (S1) we see that the Software contract `remote_sw` is trusted. We deduce using the assumption (A1) that the Software Developer `developer_sw` and the Root Owner `owner_sw` associated with the Software contract `remote_sw` are trusted. Using single function verification (see Section 4), we can show that only the Software Developer of a Software contract can add a BHS contract to their Software contract and only the Root Owner can change the Software Developer. Thus, a BHS contract associated with a Software contract is endorsed by the Software contract's Root Owner and by the Software Developer associated with the Software contract. Because of this and because the Software contract `remote_sw` endorses the given BHS contract `bhs`, we conclude that the Software Developer `developer_sw` endorses the BHS contract `bhs`.

By assumption (A2), we infer that the Software Maintainer `developer_bhs` and the Root Owner `owner_bhs` associated with the BHS contract `bhs` are trusted by the Software Developer `developer_sw`, and that `developer_sw` endorses what the Software Maintainer `developer_bhs` publishes. Using single function verification (see Section 4), we can show that only the Software Maintainer of a BHS contract can publish hashes and only the Root Owner can change the Software Maintainer. Because the hash $H(b)$ is registered with the BHS contract `bhs` (S3), $H(b)$ must have been published by the Software Maintainer `developer_bhs` (the specific Software Maintainer might have changed if the Root Owner replaced the Software Maintainer). Being published by Software

Maintainer `developer_bhs`, the hash $H(b)$ is endorsed by the Software Developer `developer_sw` (A3). Assuming that H is a one-way function, we deduce that the Software Developer `developer_sw` endorses the binary b. □

While writing this proof, we noticed the requirement to explicitly state the assumptions listed at the beginning of the proof. Although they might be intuitive, they need to be made explicit for formal proofs. For the future, we plan to use the insights from this manual proof in an approach where we model our application in a more powerful formalism, which allows high-level temporal specification and proofs (e.g., TLA+ [5]). In order to use lower-level proofs in that formalism, we will need to develop a methodology to integrate or translate pre- and postconditions of smart contract functions into it.

6 PRE-DEFINED ANTI-PATTERNS AND VULNERABILITIES

Proving that a smart contract application conforms to a user-defined specification is very complex. However, the most notorious attacks on smart contracts did not happen due to application-specific errors, but due to non-specific vulnerabilities, like neglecting parameter sanitising, or allowing reentrancy. Many common vulnerabilities have been identified, and a variety of tools has been developed to statically analyse smart contracts as to whether they contain any such weaknesses. Depending on the underlying methodology, such static analysis tools can be much faster than tools which prove arbitrary properties. Furthermore, they do not require the user to write specification. We apply two such tools, Securify and Mythril, to the Palinodia smart contracts in order to find out whether the tools report any vulnerabilities.

Securify. Securify[2] is a tool for static analysis and vulnerability detection. It specifies a set of violation and compliance patterns (as opposed to tools which only define violation patterns) and checks whether a contract fulfils a pattern through building a dependency graph from the byte code. For our application, the tool reports 107 violations, 18 of which are reported as “critical”. A further 15 violations are reported to be of “high” severity. The “critical” violations are all of the type “Unrestricted write to storage”. Upon manual inspection, it turns out that in all cases the write is actually restricted by a `require` statement invoking the identity management developed specifically for this project, making the flag a false positive. The “high” severity warnings concern unhandled exceptions and uninitialised state variables. Both warning types had no implication for the safety of the Palinodia contracts.

Mythril. Mythril[3] is an open tool for vulnerability detection, using symbolic execution and taint analysis. We run Mythril on the three contracts, with no timeout. The tool reports two possible assert violations and two possible integer underflows. Upon manual inspection, all reported errors turn out to be false positives.

7 CONCLUSION AND OUTLOOK

Due to the public, decentralised nature of smart contracts deployed on a platform like Ethereum, access control is a central theme of the application’s high-level security property as well as on the single-function level. The case study provides insights on the grand challenge of verifying the correctness of smart contract applications as follows. We successfully carried out a correctness proof on application level where the more complex temporal high-level properties are proven by using the correctness proofs from the single-function level. However, the proof was done manually: In our experience, applying existing tools for this task is far from easy. We are planning to further investigate this. Furthermore, we are planning an approach where we model a smart contract application in a different formalism, where high-level temporal correctness properties can be specified and proven more naturally. Specification and proofs on the level of single functions must be translated into this formalism in order to establish a connection between the model and the implementation.

Non-interactive static analysis tools can be valuable for quickly detecting possible security issues while developing smart contracts. While in our case no errors were detected by using such tools, we do not wish to generalise from a single case study.

We believe that our case study is a step towards getting access control right in smart contracts, and hope that it will motivate further research on modelling, specification, and verification of smart contract access control.

[2]https://github.com/eth-sri/securify2
[3]https://github.com/ConsenSys/mythril

ACKNOWLEDGMENTS

This work was supported by funding of the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL). We would like to thank Hannes Hartenstein, Martina Zitterbart, and the anonymous reviewers for their valuable feedback and comments.

REFERENCES

[1] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. 2021. A Survey on Formal Verification for Solidity Smart Contracts *(ACSW ’21)*. Association for Computing Machinery, New York, NY, USA, Article 3.
[2] Akos Hajdu and Dejan Jovanovic. 2020. solc-verify: A Modular Verifier for Solidity Smart Contracts. In *Verified Software. Theories, Tools, and Experiments*, Supratik Chakraborty and Jorge A. Navas (Eds.). LNCS, Vol. 12301. Springer.
[3] Michael A Harrison, Walter L Ruzzo, and Jeffrey D Ullman. 1976. Protection in Operating Systems. *Commun. ACM* 19, 8 (1976).
[4] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE CSF*. IEEE.
[5] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994).
[6] Alexander Mense and Markus Flatscher. 2018. Security Vulnerabilities in Ethereum Smart Contracts *(iiWAS2018)*. Association for Computing Machinery, New York, NY, USA.
[7] Daniel Perez and Benjamin Livshits. 2020. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. *arXiv:1902.06710 [cs]* (Oct. 2020).
[8] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020). IEEE.
[9] Frederik Reiche, Jonas Schiffl, Bernhard Beckert, Robert Heinrich, and Ralf Reussner. 2021. *Modeling and Verifying Access Control for Ethereum Smart Contracts*. Technical Report 1000129607. KITopen.
[10] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart Contract: Attacks and Protections. *IEEE Access* 8 (2020).
[11] Oliver Stengele, Andreas Baumeister, Pascal Birnstill, and Hannes Hartenstein. 2019. Access Control for Binary Integrity Protection using Ethereum *(SACMAT ’19)*. Association for Computing Machinery.
[12] Oliver Stengele, Jan Droll, and Hannes Hartenstein. 2020. Practical Trade-Offs in Integrity Protection for Binaries via Ethereum. In *Proceedings of the 21st International Middleware Conference Demos and Posters*.