Modeling and Enforcing Access Control Policies for Smart Contracts

Jan-Philipp Töberg
Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
jtoeberg@techfak.uni-bielefeld.de

Jonas Schiffl
Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
jonas.schiffl@kit.edu

Frederik Reiche
Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
frederik.reiche@kit.edu

Bernhard Beckert
Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
bernhard.beckert@kit.edu

Robert Heinrich
Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
robert.heinrich@kit.edu

Ralf Reussner
Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
ralf.reussner@kit.edu

***Abstract*—Ethereum smart contracts expose their functions to an untrusted network. Therefore, access control is of utmost importance. Nevertheless, many smart contracts have suffered exploits due to improper design or implementation of access control policies.**

In this work, we propose an approach for modeling role-based access control policies for Ethereum smart contracts on the architecture level, and we describe a process for ensuring that the implementation is correct w.r.t. that model. We achieve this through a combination of code generation, formal verification and static code analysis. Additionally, we provide an argument for the correctness of our approach and demonstrate its feasibility by detecting manually introduced violations in a case study.

***Index Terms*—Smart Contracts, Access Control, Ethereum**

I. Introduction

Smart contracts are programs which work in conjunction with a distributed ledger, managing resources and transactions. In recent years, the applications built with smart contracts have been getting more complex, with the most prominent platform being Ethereum.

The growing complexity of applications entails more complex access control policies. This is especially true for a public blockchain environment, where every function can be called by everybody in the network as a matter of principle. Access control failures have long been recognized as an important source of exploits.[1] While some research has been done in the area and better practices continue to be adopted, attacks which exploit flawed access control continue to happen [1].

Therefore, we present an approach to achieve correct-by-construction access control for Ethereum smart contracts.

An established approach to handle complexity in software systems is selecting an appropriate level of abstraction with information sufficient for the purpose at hand. Based on this observation, we define access control policies for smart contracts on the software architecture level using role-based access control (RBAC) as the underlying access control model. This enables application developers to either focus on the design of the architecture and access policies or the implementation of the smart contracts.

However, when a developer implements smart contracts or changes existing ones, errors can arise that violate the specified access policies. In order to ensure that the application correctly implements the policies, we design a development process which employs a combination of code generation and formal verification. From the architecture model, we derive a source code skeleton consisting of contracts with state variables and function stubs. Access to functions is restricted by Solidity modifiers and a generated access control smart contract. Furthermore, functions and formal specification are generated as well, to enforce the modeled access control policy. After the implementation of the function stubs is finished, the developer will analyze it using the formal verification tool SOLC-VERIFY [2] and the static analysis framework SLITHER [3]. If the tools do not indicate any violation, then the implementation accords with the access control policy defined on the architecture level. Otherwise, the output of the tools helps the developer to find the parts of the implementation where violations of the policy are possible.

To summarize, our contributions are as follows. First, we define a formal model to describe role-based access control policies for smart contracts at the architecture level. Second, we define what constitutes a correct implementation of these policies on the source code level. Third, we develop a meta-model for the description of Solidity applications with access restriction based on RBAC policies. Fourth, we implement a code generator which produces Solidity smart contract stubs with formal annotations. Lastly, we define an iterative process of modeling, implementation, and formal verification. The result of this process is an implementation which is guaranteed to be correct with respect to the initial RBAC policy.

This paper is structured as follows. In Section II, we present

[1] Cf. https://swcregistry.io/

related work. In Section III, we introduce relevant aspects of the Solidity programming language, the concept of RBAC used in our approach, and our running example. Going forward, we present our main approach in Section IV, and evaluate it in Section V. Section VI concludes this paper.

II. Related Work

To achieve our goal of correct-by-construction access control for Solidity smart contracts, we combine the techniques of model-driven software development (MDSD) and formal verification.

The importance of access control in the context of blockchain applications is underlined by a lot of recent research. Several approaches deal with implementing access control for the functionality provided by Ethereum applications, such as Chatterjee et al. [4]. In contrast, while our approach also deals with access to functions, our main focus is access to smart contract state.

Several approaches exist that use MDSD to model smart contracts and access control, including its policies, and generate the resulting Solidity source code.

The approach Caterpillar by López–Pintado et al. [5] is an execution engine for smart contract applications. This approach allows developers to send a model of their application written in the Business Process Model and Notation language through a REST API to be compiled and deployed on the Ethereum blockchain.

To model access control for smart contracts, BlockchainStudio [6] extends Caterpillar with model elements for describing RBAC policies like organizations, users and roles. Additionally, each function is extended by defining which roles are allowed to access it. These permissions are checked at runtime using an additional smart contract that maps addresses to users and roles.

A framework with similarly extensive capabilities to Caterpillar is FSolidM by Mavridou and Laszka [7], which allows for the modeling of smart contracts using transition-based finite state machines (FMSs). Based on such an FSM, the approach automatically generates Solidity source code. Additionally, it allows for the addition of design patterns. One of these is a simple ownership-based access control, which defines a single address for each contract as the owner, who has additional permissions to access vulnerable parts of the contract. Our approach focuses explicitly on access control and allows for much richer access control policy specification.

There are several approaches applying the analysis and verification of smart contracts. The VeriSolid extension by Mavridou et al. [8] introduces formal verification for the FSolidM framework by Mavridou and Laszka [7]. The newly introduced formal verification is done by employing model checking on the transition-based models to reason about deadlock freedom, liveness and safety.

Alhabardi et al. [9] use the interactive theorem prover Agda to verify Bitcoin smart contracts. This verification is applied to smart contracts that focus on the security property of access control. Similarly, Schiffl et al. [10] rely on the formal verification capabilities of SOLC-VERIFY [2] to verify the correctness and temporal properties of the Ethereum access control system Palinodia [11].

Our approach is similar in that it uses formal verification tools. However, instead of manually specifying the desired property, the focus on access control allows us to use a more suitable layer of abstraction for the description of the policies. Our combination of MDSD and formal verification implements a separation of concerns: the descriptions of policies happen on a more abstract level, while implementation and verification take place on the source code level.

Combining MDSD and formal verification to reason about access control for smart contracts is done in a technical report by Reiche et al. [12]. Their approach relies on the Palladio Component Model as an architectural model for describing access control policies. Based on this model, smart contracts are generated and formally verified using SOLC-VERIFY [2]. Using the Palladio Component Model introduces limitations regarding complex data types and their verification. Additionally, the work of Reiche et al. uses an ad-hoc approach for the generation. We complement this approach by providing a formal basis for our transformations and generations. Furthermore, our work also considers indirect access to state variables, as opposed to just direct write access.

III. Preliminaries

A. Solidity

A prominent platform for smart contracts is Ethereum, a permissionless blockchain technology. For the development of smart contracts, Ethereum provides developers with a Turing complete instruction set in the form of the Ethereum Virtual Machine (EVM). Before a smart contract is deployed, it is compiled into low-level, stack-based bytecode that is executed on the EVM. High-level programming languages for Ethereum include Solidity and Vyper [13]. For our approach, we target the Solidity programming language, because it is the most widely used higher-level language for Ethereum.

Solidity is a "contract-oriented" programming language; the overall structure of a Solidity application consists of contracts, state variables, and functions. Contracts are similar to objects in object-oriented programming languages; Solidity also offers interfaces and inheritance.

In Ethereum, there are external accounts (representing actual persons or entities) and contract accounts (representing smart contracts on the Ethereum blockchain). For our approach, there is no need to differentiate between the two. In the Solidity programming language, accounts are identified by the *address* datatype. The values for this datatype are unique 160 bit integers, allowing for an unambiguous identification of entities.

Every smart contract has its own state, which is defined in the form of state variables. Built-in data types include different integer types, booleans and strings as well as arrays, structs and mapping types. The state of a contract also includes its balance in Ether, the currency of the network.

Access to a contract's state is managed by its functions. Functions in Ethereum are transactional, i.e., they either succeed or revert completely. In the context of Ethereum, a transaction is a function call which potentially changes the state.

The business logic of a smart contract often depends on the current state of the network environment, e.g., the caller of a function, the time, the current block number, or the amount of currency that was transferred in a call. Within a smart contract, these are accessible via the *msg* and *block* fields (e.g., *msg.sender*, *block.timestamp*, *block.number*, and *msg.value*).

Functions can be private (only callable from within a contract) or public. Public functions are exposed to the entire Ethereum network and can be called by any account. Therefore, in a sense, a Solidity function has to implement its own access control – if some functionality should only be accessible to a subset of users, then a function needs to check at runtime whether the caller is in that subset.

One way to achieve this is the *require* keyword, which takes a boolean condition. If the specified condition fails at runtime, the current function terminates and any changes to the contract's state are reversed to the state before the function call.

Another concept of Solidity are modifiers, which are employed to change or influence the behavior of the function they annotate. For this purpose, a new function is created with the modifier keyword instead of the function keyword. The name of a modifier can be added to the signature of any function in the contract. In the modifier's code, the sequence *_;* marks the behavior of the original function, so a modifier can be used to add behavior before or after the original function.

B. Formal Analysis Tools

Ethereum smart contracts often manage cryptocurrency or tokens representing real-world assets. Since smart contracts cannot be changed after deployment, it is of supreme importance that they work as intended and contain no bugs. To ensure this, different tools were developed to analyze the behavior and structure of smart contracts as early in the development process as possible. By doing so, developers are supported in creating safe and secure smart contracts, and the possibilities for attacks and exploitation are reduced.

For our approach, we rely on the capabilities of SLITHER [3], a static analysis framework, and SOLC-VERIFY [2], a tool for the formal verification of Solidity smart contracts.

The static analysis framework SLITHER by Feist, Grieco, and Groce [3] allows for the detection of vulnerabilities and possible code optimizations as well as supporting developers with understanding the source code. For this, it takes the abstract syntax tree created by the Solidity compiler and transforms it into a hierarchical representation that supports object-oriented access through its public API. This representation is utilized by different implemented extensions to either analyze the smart contract for the presence of certain vulnerabilities like reentrancy, or summarize its structure for a better understanding with a call or inheritance graph. Additionally, the framework provides developers with a public python API enabling the development of custom extensions that implement new types of analysis or contract summary.

The formal verification tool SOLC-VERIFY by Hajdu and Jovanović [2] analyzes the correctness of Solidity smart contracts on a function-to-function basis. For this purpose, it provides an annotation language that allows to add formal specifications in the form of comments to Solidity smart contracts. These take the form of contract-level invariants, functional pre- and postconditions as well as loop invariants. The specification language is a combination of first-order logic and Solidity. For example, the *require* keyword introduced in Section III-A is taken into consideration as an additional precondition.

For our work, the most relevant aspect of SOLC-VERIFY's specification language are frame conditions (or *modifies* clauses). These are part of a function contracts and specify which part of the state may be modified by the function. A successful proof of a function's contract means that the function can modify at most those state variables mentioned in the modifies clause.

C. Role-Based Access Control (RBAC)

Access control focuses on stopping unwarranted entities from accessing data or functionality they are not allowed to see, use, or change. An access control request consists of a *subject*, which can be a user or anything that wants to access a resource, and an *object*, which is the resource or element that should be accessed. A *policy* defines the rules describing which subjects are allowed to access which objects. Each policy is an instance of an *access control model*, which formalizes the aspects that are considered in the policy [14].

Due to the public nature of permissionsless blockchains like Ethereum, read access is possible under any circumstances and cannot be restricted. Therefore, we understand access control as restricting write access.

For our approach, we employ role-based access control (RBAC), where permissions are abstracted from single entities to roles. Therefore, the subjects are handled as groups of individual entities that all possess the same permissions. These entities are represented in the Solidity programming language by the *address* data type. For the smart contract domain, we modify the formal standard model of RBAC by Sandhu et al. [15]. The resulting formal model contains three sets: *Functions*, *State Variables* and *Roles*. These sets are connected to describe policies where certain roles are permitted to access certain functionality or data.

The assignment of entities to roles happens either statically or dynamically [14]. For a static role assignment, the entities are added as a part of the model on the architectural level and the assignment does not change at runtime. A dynamic assignment on the other hand is done only at runtime on the source code level, allowing for more flexibility. This is especially relevant for permissionless blockchains like Ethereum,

where not all accessing entities are known to the software architect beforehand.

D. Secure Information Flow

Apart from modeling direct write access, our approach also considers information flow between state variables. Specifically, we handle insecure information flow as defined by Terauchi and Aiken [16]. The authors define a secure information flow as a program where the final values for the low-security variables do not depend on the initial values of the high-security variables.

In the domain of smart contracts, we define these secure information flows in correspondence with the roles from the formal model. Here, an information flow is secure iff the value of variables that must not be modified by a role does not depend on the value of variables a role can modify – neither directly via an assignment nor indirectly via effects on control flow. An insecure information flow, on the other hand, allows a role to influence the value of a state variable.

For our purposes, we use the following definition. Insecure information flow takes one of the following three forms, which are all considered in our approach:

1) *Direct*: The value of one state variable is directly assigned to another state variable.
2) *Indirect*: The value of one state variable depends on another state variable through conditions like if-then-else or *require*.
3) *Transitive*: The value of one state variable s_1 depends on another state variable s_2, which itself depends on another state variable s_3. In this case, s_1 also depends on s_3. The connection between s_1 and s_2 as well as the connection between s_2 and s_3 can be direct, indirect, or transitive.

E. Running Example: Auction

Originating from the Solidity documentation [17], our running example is based on the real-world scenario of an auction, where different entities bid on items that are for sale. In the open version from the documentation, all participants see who is bidding what amounts of money on which item.

The version we employ consists of two smart contracts, *SingleAuction* and *AuctionManagement*. By calling a publicly accessible function of the *AuctionManagement* contract, any entity can create a new auction, represented by a *SingleAuction* contract. Beside this creation, the *AuctionManagement* provides no additional functionality. In the *SingleAuction* contract, four different roles are available:

- The *seller* represents the entity responsible for the creation of the auction. This role has the permission to collect the money after the auction ended.
- The *auction manager* can shutdown all auctions in the case of an emergency (e.g. a security breach in the contract).
- Bidding is publicly available to all entities. Any bidding entity is assigned to the *bidder* role, allowing them to withdraw the money they bid before the auction has ended. The *highest bidder* is a specialization of the *bidder* role. It can only be assigned to one entity at any point in time. This entity, as the name suggests, is the one that bid the most money during the auction and is thus allowed to collect the auction's item.

IV. Enforcing RBAC Policies for Solidity Smart Contracts

This section describes our contributions. First, we describe how we model Solidity smart contracts with role-based access control policies. Next, we define what constitutes a correct implementation of such a model. Then, we describe an approach for how, given a model, an implementation can be developed which is guaranteed to fulfill that notion of correctness.

A. Formally Modeling RBAC Policies

To formally describe RBAC policies for smart contracts, we build on the formal RBAC model we introduced in Section III-C. First, we define the three basic sets of functions, state variables, and roles:

- $\mathbb{F}$: Set containing all functions
- $\mathbb{S}$: Set containing all state variables
- $\mathbb{R}$: Set containing all roles

The core of our model is the description of which roles can (1) access which functions, and (2) access which state variables. For modeling admissible access of a role to a function, we define the relation $RtoF$. The access to state variables is split into direct write access (RmS) and indirect access (RiS):

- $RtoF \subseteq \mathbb{R} \times \mathbb{F}$: $(r, f) \in RtoF$ *iff* role r is allowed to call function f.
- $RmS \subseteq \mathbb{R} \times \mathbb{S}$: $(r, s) \in RmS$ *iff* role r is allowed to modify the state variable s.
- $RiS \subseteq \mathbb{R} \times \mathbb{S}$: $(r, s) \in RiS$ *iff* role r is allowed to influence the state variable s. Each influence relation constitutes a possibly insecure information flow as introduced in Section III-D.

Furthermore, we define some additional relations, which serve as auxiliary specification for code generation and interpretation of the verification tool results:

- $FtoS \subseteq \mathbb{F} \times \mathbb{S}$: $(f, s) \in FtoS$ *iff* function f is allowed to write state variable s – either directly or transitively through its called functions.
- $StoS \subseteq \mathbb{S} \times \mathbb{S}$: $(s_1, s_2) \in StoS$ *iff* an information flow from state variable s_1 to state variable s_2 exists
- $FtoF \subseteq \mathbb{F} \times \mathbb{F}$: $(f_1, f_2) \in FtoF$ *iff* function f_1 is allowed to call function f_2. This also includes transitive function calls.

Lastly, we add a subset of the role *authorization constraints* defined in the Generalized Model for RBAC by Ben Fadhel, Bianculli and Briand [18]. These constraints add connections between roles like hierarchy, mutual exclusion or prerequisites. Additionally, the amount of entities assigned to a specific role can be limited and a temporal or boolean context for the access can be specified.

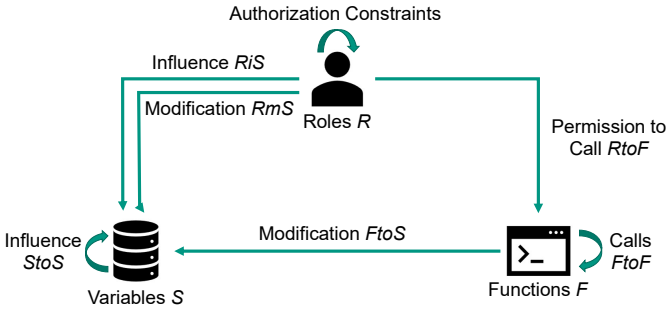

Fig. 1. Visual representation of the formal model we employ as a foundation for describing RBAC policies for smart contracts.

Figure 1 gives a complete visual overview of this formal model. An application of the model on the auction example from Section III-E is given in Figure 2, where the different sets and relations are represented by visual elements.

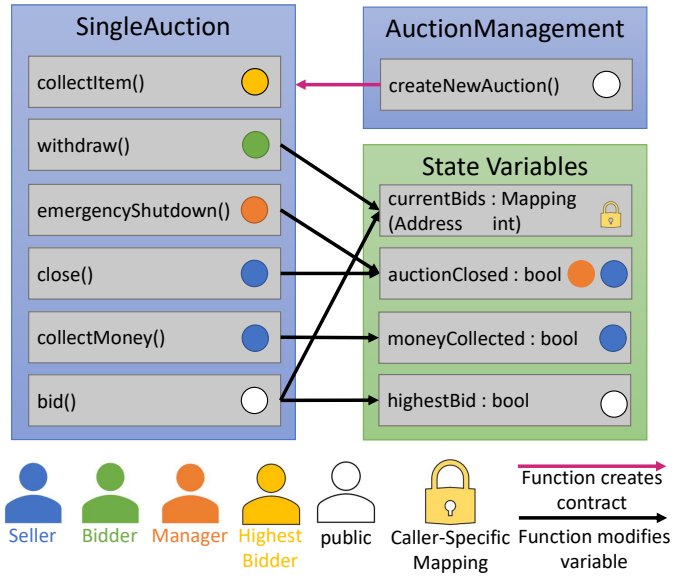

Fig. 2. Visual representation of the *SingleAuction* and *AuctionManagement* smart contracts from our running example. The colored dots symbolize which of the roles is permitted to call the function or modify the state variable. The lock represents a caller-specific mapping, which we define in Section IV-C.

B. Correctness of Smart Contracts w.r.t. a Model

This section gives a brief overview of what it means for the implementation of a smart contract application to be correct w.r.t. a model as described above.

We assume that, for every $f \in \mathbb{F}$ and every $s \in \mathbb{S}$, there is a corresponding function or state variable, respectively. For roles, there is no direct equivalent in Solidity. Instead, individuals are represented by Ethereum addresses and roles correspond to collections of addresses. To represent this in the code, the application is assumed to provide a method to check whether a given address has a given role at a certain point in time.

In our approach, these assumptions are ensured by the code generation; they hold by construction.

Then, to be correct, an implementation must satisfy the following conditions:

- an implementation is correctly handling access to functions if, whenever a function f is called by an address, then either the address has a role r that is allowed to call f, i.e., $(r, f) \in RtoF$ or the function reverts
- an implementation is correctly handling direct write access if an address can invoke a transaction in such a way that the value of a state variable s is changed by that transaction only if that address has a role r such that $(r, s) \in RmS$
- an implementation is correctly handling indirect access if an address can effect the change of any state variable that s depends on only if it has a role r such that $(r, s) \in RiS$. A variable x depends on a variable y iff, given two states which are equal up to the value of y, there is a sequence of function calls that, when applied to the two states, results in states with different values of x.

We deem an implementation correct if it fulfills all three of these requirements.

C. Modeling RBAC Policies on the Architectural Level

We represent the formal model on the architectural level by employing the Eclipse Modeling Framework (EMF) to create a new metamodel. The resulting *AccessControlMetamodel (ACM)* is an instance of the Ecore meta-metamodel and is instantiated to create a concrete model which represents the RBAC policies for a specific use case. The ACM consists of two packages: The *SmartContractModel* package provides elements to describe the smart contract (e.g. functions and state variables) whereas the *AccessControlSystem* package focuses on all elements representing the RBAC policies (e.g. roles and the connecting relations). In addition to the aspects described in the formal model, the ACM also allows for modeling access to storage locations of a *mapping* data structure as well as access to a contract's balance. This covers the balance of the modeled contract as well as the contract of the function caller (represented by Solidity's *msg.sender* keyword).

To reason about the soundness of the model instances, we define explicit constraints using the OCL language. These 15 constraints include, for example, a check that the cardinality of any role is described with a positive integer greater then zero or that the two roles selected as mutually exclusive are not the same. By verifying these constraints, we prevent syntactic errors from occurring in the model instances.

Similarly, to verify the soundness of the modeled RBAC policies, we define a list of properties that the ACM instances need to fulfill. Otherwise, the supporting relations from Section IV-A may not hold and the generator could not create the foundation for a correct implementation.

- If a role r is allowed to call a function f ($(r, f) \in RtoF$) but not allowed to **modify** a state variable s ($(r, s) \notin RmS$), f is restricted from modifying s ($(f, s) \notin FtoS$). Also, all functions called by f are restricted from modifying s.

- If a role r is allowed to call a function f ($(r, f) \in RtoF$) but not allowed to **influence** a state variable s ($(r, s) \notin RiS$), f is restricted from influencing s ($(f, s) \notin FtoS$). Additionally, all functions called by f are restricted from influencing s and all state variables modified by f are not allowed to influence s
- If a role r is allowed to modify a variable s_1 ($(r, s_1) \in RmS$) but is not allowed to influence another state variable s_2 ($(r, s_2) \notin RiS$), s_1 is banned from influencing s_2 ($(s_1, s_2) \notin StoS$)
- If a role r is allowed to call function f ($(r, f) \in RtoF$), it also needs permission to call all functions which f is allowed to call.

D. Generation of Solidity Smart Contracts

After creating the ACM for representing RBAC policies on the architectural level, we define and develop an approach for enforcing these policies on the source code level. To achieve this goal, we implement a generator capable of translating the instances of the ACM into Solidity smart contracts. This generator, as well as the metamodel, is publicly available in our GitHub repository[2]. The generated smart contracts contain the modeled state variables and function stubs. Since the function's behavior is not modeled in the architecture, the function's body is left empty. To enforce the correctness of the RBAC policies, the generator generates a smart contract to perform the access control, modifiers to restrict the access to functions and annotations for the SOLC-VERIFY tool to enable verification of the access of functions to state variables.

a) Generation of the Access Control Smart Contract: To dynamically manage the assignments of Solidity addresses to roles, we generate an additional access control smart contract. The result of the generation for the running example is displayed in Listing 1. The roles are represented by an enumeration (cf. line 1). The assignment is stored as a nested mapping (cf. line 2), which maps each address to a second mapping that connects each available role with a boolean value. Changes to the role assignment are done by changing a specific mapping location. The functions for accessing and verifying the mapping are generated as well. The $checkAccess$ function, for example, begins in line 4 and is employed to analyze the entity-to-role assignment at runtime. In general, the complete access control contract is fully functional and must not be modified after its initial generation. However, the software developer needs to manually implement the dynamic entity-to-role assignment that is enforced at runtime. This is done by calling generated functions in the access control contract in the implementation of the partly generated smart contracts.

b) Generation of Solidity Modifiers: In order for the access control policies to be fulfilled, access to functions has to be restricted to entities that have a certain role. We achieve this using Solidity modifiers (cf. Section III-A). A modifier adapts a function's behavior. For our approach, we generate a modifier for each function f. The modifiers check if the calling address is assigned to any of the permitted roles defined by $(r, f) \in RtoF$. To check this role assignment, the $checkAccess$ function of the access control contract is called. For example, in Listing 2 the *onlySeller* modifier defined in line 11ff is employed in line 6 to restrict access to the *close* function to entities that are assigned to the *seller* role.

[2]https://github.com/KASTEL-CSSDA/SolidityAccessControlEnforcement

```
1  enum Roles { SELLER, BIDDER, HIGHEST_BIDDER,
       MANAGER, ADMIN }
2  mapping(address => mapping(Roles => bool))
       private roleAssignment;
3
4  function checkAccess(address entity, Roles role)
       public view returns(bool result) {
5    return roleAssignment[entity][role];
6  }
```

Listing 1. Excerpt from the generated access control smart contract for the running example from Section III-E.

c) Generation of SOLC-VERIFY *Annotations:* In addition to the Solidity code stubs, we generate annotations in the specification language of the SOLC-VERIFY tool. Specifically, we create frame conditions: For any function f, we generate one modifies clause mentioning the state variable s for each $(f, s) \in FtoS$. SOLC-VERIFY either provides a proof that a function modifies at most the state variables in its frame condition, or the proof fails and the tool reports the possible violations. An example frame condition is provided in line 4 of Listing 2, where the *close* function is permitted to modify the *auctionClosed* state variable.

```
1  bool private auctionClosed;
2  mapping(address => uint) private currentBids;
3
4  /// @notice modifies auctionClosed
5  /// @notice modifies currentBids[msg.sender]
6  function close() public onlySeller {
7    auctionClosed = true;
8    currentBids[msg.sender] = msg.value;
9  }
10
11 modifier onlySeller {
12   require(accCtrl.checkAccess(msg.sender,
       AccessControl.Roles.SELLER));
13   _;
14 }
```

Listing 2. Excerpt from the generated *SingleAuction* smart contract based on the model visually represented in Figure 2.

Similarly, access to a *mapping* data structure can be restricted to only allow changes to the storage location associated with the address currently calling the function. This restriction is enforced with SOLC-VERIFY annotations, like the comment in line 5 of Listing 2. This annotation formalizes that the *currentBids* state variable may only be modified at the storage location represented by the *msg.sender*. Regarding changes to the contract's balance, we employ postconditions to compare the balance before and after the function's execution.

E. Implementation Process

Based on the presented approach, we define a process for developing an implementation which is correct w.r.t. a given RBAC model. Figure 3 shows an outline of the process. It involves two stakeholder roles, the software architect and the software developer, as well as three tools: SOLC-VERIFY [2], SLITHER [3] and our generator.

The process begins with the software architect that creates instances of the ACM as a representation of the real-world scenario. In the next step, a soundness check verifies the fulfillment of the properties and OCL constraints introduced in Section IV-C. Therefore, violations to the underlying RBAC model are detected early during the development process and are communicated back to the architect. Only when no violations occur, the generator creates the smart contract stubs annotated with formal specifications as described in Section IV-D. After the generation, the software developer manually implements the smart contract function stubs without modifying the generated elements like the modifiers or annotations.

For the verification of this implementation, SOLC-VERIFY and SLITHER are employed. As we mentioned in Section IV-D, SOLC-VERIFY formally verifies the generated modification specifiers and postconditions. Therefore, if a function f modifies a state variable s that it is not permitted to modify ($(f, s) \notin FtoS$), an error is reported. To reason about the influence relation between state variables and roles, we employ the static analysis framework SLITHER. However, the analyses incorporated in the default framework do not cover all aspects of the influence relation mentioned in Section III-D. This prompted us to implement an extension that returns a comprehensive overview where all influencing variables for each state variable are shown. This includes transitive influence as well by calculating the transitive closure. The implementation for the SLITHER extension is also available in our repository. After the influence overview is created, a stakeholder needs to manually check each connection between two state variables s_1 and s_2. Due to the overapproximation introduced by calculating the transitive closure, it first needs to be checked if the connection is a false positive. Every influence connection that cannot occur at runtime but is introduced by our overapproximation qualifies as a false positive and is excluded from further processing. If it is a true positive, it is checked whether an element representing the influence connecting between s_1 and s_2 exists in the model ($(s_1, s_2) \in StoS$). If such a model element exists, no violations occur. However, if no such element exists, the implementation may handle indirect access incorrectly w.r.t the model, as we explained in section IV-B. So the stakeholder needs to check whether any role gains unwarranted influence access through this detected influence connection between two state variables. If no role gains access, a shortcoming in the ACM instances is detected. Otherwise, the gained access violates the modeled RBAC policies.

If such an access violation is detected in the implementation by either tool, it needs to be communicated back to the stakeholders. When the results are communicated to the software developer, they can be used to correct the implementation to follow the designed access control policies. On the other hand, when the results are communicated to the software architect, the mismatch between architecture and implementation are made explicit and implications on the security of the total system can be evaluated. Therefore, either the source code or the architectural models are adapted to prevent the violations from occurring. To connect the verification results with the model or implementation elements, we propose to employ a correspondence model, as explained in [19]. However, if no violations are found, the smart contracts are ready for deployment.

F. Correctness of the Approach

The goal of our process outlined above is to stop unwarranted entities from accessing functionality and modifying state variables. In section IV-B, we gave a definition of what constitutes a correct implementation w.r.t. a formal model of an application. Here, we argue why the process described above results in an implementation which is correct in that sense.

The functions and state variables as well as the $checkAccess$ method, which tests whether a given address has a given role, are all provided by our code generator.

Access to functions is handled by the generated modifiers and their inclusion in the corresponding function headers as described in Section IV-D. These modifiers ensure that an address which does not have a role that allows it to call a function, as specified by $RtoF$, will only cause the function to revert with an error message. This is the implementation of access being denied.

Direct write access is first addressed on the architecture level, where a model is only accepted by our modeling tool if a role which may not access a variable may also not access any function which can modify that variable. On the source code level, this is enforced by a combination of function modifiers and verified frame conditions. Access to functions is restricted through modifiers. Since SOLC-VERIFY is sound, a successful verification of the generated frame conditions guarantees that the $FtoS$ relation is correctly implemented. Since in Solidity there is no way of changing a state variable except through function calls, we conclude that a resulting implementation adheres to our definition of correctness.

As for indirect access, the argument is largely analogous to the one for direct write access. SLITHER analyzes which dependencies between variables (in the sense of the definition in section IV-B) exist, and access control to functions prevents addresses lacking the necessary roles from modifying state variables on which the critical variables depend.

In contrast to SOLC-VERIFY, SLITHER does not give a soundness guarantee. Therefore, in theory, it is possible that an implementation still contains information flows which are forbidden by the model even if SLITHER does not report them. In practice, we found no such examples, however.

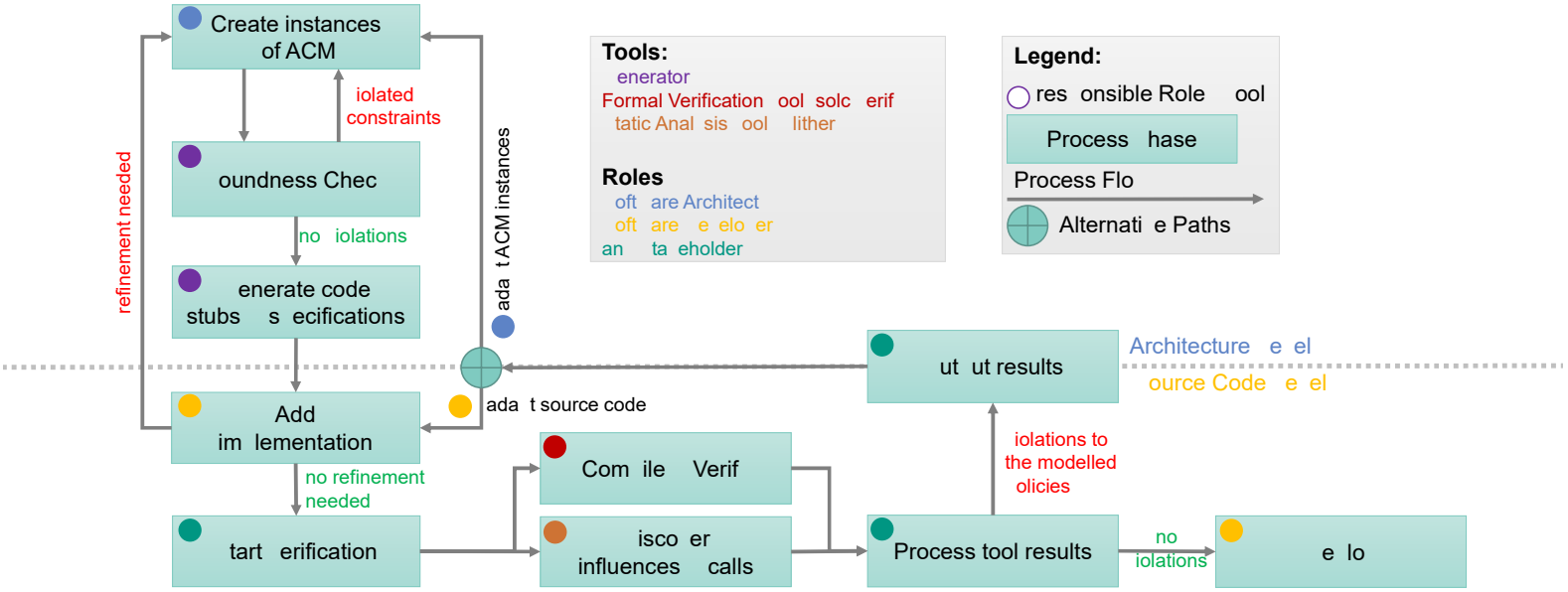

Fig. 3. The process envisioned for developers and architects using the presented approach to enforce their smart contract RBAC policies.

V. Evaluation

To empirically demonstrate the feasibility of our approach, we manually implement three real-world scenarios and their access control by following the process outlined in Figure 3. For each scenario, we model the roles, functions and state variables on the architectural level with the model from Section IV-C. This model is based on the system's description by the original developers, where we fill in any existing gaps. During this implementation, we add violations for the different requirements described as a foundation for a correct implementation in Section IV-B. By showing how our approach is able to identify these violations we underline its capability to assess the correctness of an implementation w.r.t. the modeled RBAC policies. The resulting implementations as well as the model instances are available in our GitHub repository.

The three scenarios we employ for this case study vary in complexity. However, all of them benefit from a correct enforcement of access control. Due to the size of the Palindia [11] and Augur [20] scenario, we will focus on the third scenario, Fizzy [21], here. Fizzy was an automatic flight delay insurance system developed by the insurance company AXA, allowing for a tamper-proof payment if a delayed flight occurs. It consisted of two contracts, *InsuranceManagement* and *Insurance*, with only two roles, the *insurant* and the *insurance company*. A visual summary can be examined in Figure 4.

As we mentioned in Section IV-E, all four properties for sound model instances introduced in Section IV-C are analyzed once on the architectural level before anything is generated. If, for example, the *insurant* in the Fizzy scenario is modeled to start the payout process, this violation is communicated to the software architect before the generation continues. On the source code level, violations to the correctness of the

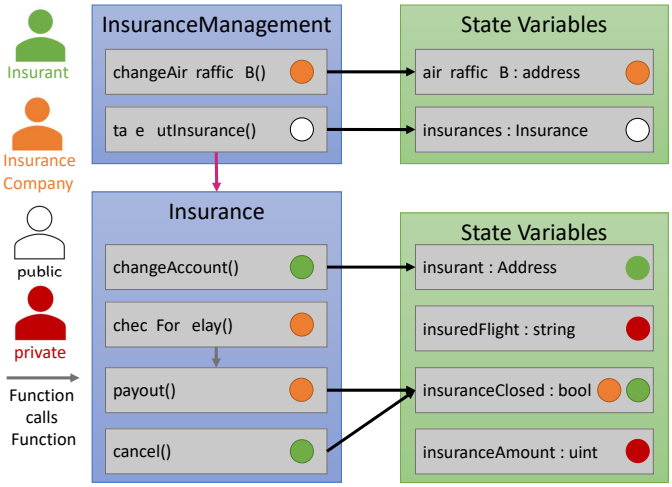

Fig. 4. Visual representation of the Fizzy scenario [21]. It relies on the same visual elements introduced in Figure 2 to describe our running example.

implementation described in Section IV-B are detected by SOLC-VERIFY and SLITHER. Incorrectly handling direct write access allows a role r to illegally modify a state variable s through function f. For the Fizzy scenario we demonstrate this violation by setting the $insurant$ state variable to zero in line 7 of Listing 3. This access allows the *insurance company* role to illegally modify the $insurant$ variable. However, since no modification specifier is generated for the modification done by the $payout$ function to the $insurant$ variable ($(payout, insurant) \notin FtoS$), this violation is detected by SOLC-VERIFY.

Incorrectly handling indirect access results in insecure information flows between state variables that are identified by our SLITHER extension. Currently, the results returned

```
1  address private insurant;
2  bool public insuranceClosed;
3
4  /// @notice modifies insuranceClosed
5  function payout() internal onlyInsuranceCompany {
6    insuranceClosed = true;
7    insurant = address(0);
8  }
9
10 /// @notice modifies insurant
11 function changeAccount(address newInsurant)
       external onlyInsurant {
12   require(!insuranceClosed);
13   insurant = newInsurant;
14 }
```

Listing 3. Excerpt from the generated *Insurance* smart contract based on the Fizzy scenario visually represented in Figure 4.

by this extension are manually analyzed by a stakeholder to find violations that would allow a role to illegally influence a state variable or call a function. As an example violation, we manually introduce an influence relation between the $insurant$ and the $insuranceClosed$ variable ($(insuranceClosed, insurant) \in StoS$) by making changes to the $insurant$ variable depend on the current state of the insurance contract in line 12 of Listing 3. However, this influence relation allows the *insurance company* role to illegally influence the $insurant$ state variable through the $changeAccount$ function ($(insurance\ company, insuranceClosed) \in RmS$ and $(insurance\ company, insurant) \notin RiS$).

With the implementation of these three use cases, we demonstrate the feasibility and applicability of our approach for small- to medium-sized real-world scenarios. In our implemented cases, all manually introduced violations to the RBAC policies were detected.

VI. Conclusion and Future Work

We presented an approach for modeling Ethereum smart contracts with role-based access control on the architecture level. We also implemented a code generator, and showed how a developer can use formal verification tools to ensure that the final smart contract correctly implements the architecture model. Additionally, we extended the static analysis framework SLITHER to reason about information flows between state variables.

We formally defined the underlying RBAC model and what constitutes a correct implementation w.r.t. this model. By following the presented process, we reasoned about the correctness of our approach in stopping unwarranted entities from accessing state variables.

Additionally, we executed an empirical evaluation regarding the feasibility of our approach with a case study. This case study is made up of three different, real-world scenarios which we implemented. By manually adding violations to the model and the implementation, which were identified by our approach, we outline the capabilities of our approach.

Currently, the developer still has to interpret the output of SOLC-VERIFY and SLITHER in order to identify which part of the access control policy is violated by the implementation. The detected violations also have to be manually integrated to the architectural model to identify errors arising from them to the whole system. Therefore, in the future, we plan to add functionality to provide a unified output of violations in relation to the access control policies and therefore abstract from the output of SOLC-VERIFY and SLITHER. In addition, we plan to use this unified output to integrate the violations into the architectural model by application of transformation rules. This approach enables the application of further analysis with the policies that are aligned with the source code.

While our approach is specific for Solidity smart contracts, the basic necessities for access control are the same for other Ethereum programming languages (e.g., Vyper [3]) and even other public blockchain networks. Similarly, the formal verification relies on concepts that allow for different tools with similar functionalities to be employed instead of being limited to SOLC-VERIFY and SLITHER.

Acknowledgment

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.01 Methods for Engineering Secure Systems)

References

[1] B. C. Gupta, N. Kumar, A. Handa, and S. K. Shukla, "An Insecurity Study of Ethereum Smart Contracts," in 10th Int. Conf. on Security, Privacy, and Applied Cryptography Engineering (SPACE 2020), Kolkata, India, 2020, pp. 188–207.

[2] Á. Hajdu and D. Jovanović, "solc-verify: A Modular Verifier for Solidity Smart Contracts," in 11th Int. Conf. on Verified Softw., Theories, Tools and Experiments (VSTTE 2019), New York City, NY, USA, 2019, pp. 161–179.

[3] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in 2nd Int. Workshop on Emerging Trends in Softw. Eng. for Blockchain (WETSEB 19), Montreal, QC, Canada, 2019, pp. 8–15.

[4] Chatterjee, A., Pitroda, Y., Parmar, M. (2020). Dynamic Role-Based Access Control for Decentralized Applications. In: Chen, Z., Cui, L., Palanisamy, B., Zhang, LJ. (eds) Blockchain – ICBC 2020. ICBC 2020. Lecture Notes in Computer Science(), vol 12404. Springer, Cham. https://doi.org/10.1007/978-3-030-59638-5_13

[5] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, and A. Ponomarev, "Caterpillar: A business process execution engine on the Ethereum blockchain," Software: Practice and Experience, vol. 49, no. 7, pp. 1162–1193, 2019, doi: 10.1002/spe.2702.

[6] L. Mercenne, K.-L. Brousmiche, and E. B. Hamida, "Blockchain Studio: A Role-Based Business Workflows Management System," in 9th IEEE Annu. Inf. Technol., Electronics and Mobile Communication Conf. (IEEE IEMCON 2018), Vancouver, BC, Canada, 2018, pp. 1215–1220.

[7] A. Mavridou and A. Laszka, "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach," in 22nd Int. Conf. on Financial Cryptography and Data Security (FC 2018), Nieuwpoort, Curaçao, 2018, pp. 523–540.

[8] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-Design Smart Contracts for Ethereum," in 23rd Int. Conf. on Financial Cryptography and Data Security (FC 2019), Frigate Bay, St. Kitts and Nevis, 2019, pp. 446–465.

[9] F. F. Alhabardi, A. Beckmann, B. Lazar, and A. Setzer, "Verification of Bitcoin's Smart Contracts in Agda using Weakest Preconditions for Access Control," 2022.

[3] https://vyper.readthedocs.io/

[10] J. Schiffl, M. Grundmann, M. Leinweber, O. Stengele, S. Friebe, and B. Beckert, “Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control,” in 26th ACM Symp. on Access Control Models and Technologies (SACMAT’21), Virtual Event, Spain, 2021, pp. 125–130.
[11] O. Stengele, A. Baumeister, P. Birnstill, and H. Hartenstein, “Access Control for Binary Integrity Protection using Ethereum,” in 24th ACM Symp. on Access Control Models and Technologies (SACMAT’19), Toronto, ON, Canada, 2019, pp. 3–12.
[12] F. Reiche, J. Schiffl, B. Beckert, R. Heinrich, and R. Reussner, “Modeling and Verifying Access Control for Ethereum Smart Contracts,” Inst. of Inf. Secur. and Dependability (KASTEL), Karlsruhe, Germany, Tech. Rep, 2021, https://publikationen.bibliothek.kit.edu/1000129607 (accessed: May 03, 2022).
[13] D. Harz and W. Knottenbelt, “Towards Safer Smart Contracts: A Survey of Languages and Verification Methods,” Computing Research Repository (CoRR), abs/1809.09805, 2018, https://arxiv.org/pdf/1809.09805 (accessed: May 03, 2022).
[14] P. Samarati and S. C. de Vimercati, “Access Control: Policies, Models, and Mechanisms,” in Lecture Notes in Computer Science, vol. 2171, Foundations of Security Analysis and Design: Tutorial Lectures, R. Focardi and R. Gorrieri, Eds., New York: Springer, 2001, pp. 137–196.
[15] R. Sandhu, D. Ferraiolo, and R. Kuhn, “The NIST Model for Role-Based Access Control: Towards A Unified Standard,” in 5th ACM Workshop on Role-based Access Control (RBAC’00), Berlin, Germany, 2000, pp. 47–63.
[16] T. Terauchi and A. Aiken, “Secure Information Flow as a Safety Problem,” in 12th Int. Static Analysis Symp. (SAS 2005), London, United Kingdom, 2005, pp. 352–367.
[17] A. Beregszaszi and C. Reitwiessner, ”Solidity by Example.” Solidity Documentation, https://docs.soliditylang.org/en/develop/solidity-by-example.html (accessed: May 03, 2022).
[18] A. Ben Fadhel, D. Bianculli, and L. Briand, “A comprehensive modeling framework for role-based access control policies,” Journal of Systems and Software, vol. 107, pp. 110–126, 2015, doi: 10.1016/j.jss.2015.05.015.
[19] R. Heinrich et al., “Integrating Run-Time Observations and Design Component Models for Cloud System Analysis,” in 9th Workshop on Models@run.time, Valencia, Spain, 2014, pp. 41–46.
[20] J. Peterson, J. Krug, M. Zoltu, A. K. Williams, and S. Alexander, “Augur: A Decentralized Oracle and Prediction Market Platform (v2.0),” Whitepaper, Forecast Foundation, 2021, https://github.com/AugurProject/whitepaper/releases/latest/download/augur-whitepaper-v2.pdf (accessed: May 03, 2022).
[21] L. Benichou, ”fizzy. Innovation at AXA.” Medium.com, https://laurentbenichou.medium.com/fizzy-innovation-at-axa-2304bb71e0f7 (accessed: May 03, 2022).