Modelling and Verifying Access Control for Ethereum Smart Contracts

Frederik Reiche, Jonas Schiffl, Robert Heinrich, Bernhard Beckert, Ralf Reussner

Karlsruhe Institute of Technology
`firstname.lastname@kit.edu`

Abstract. Smart contracts are programs on decentralized platforms. They provide services in the form of function calls, which are in principle visible to and callable by everyone on the network. However, smart contracts often contain some functionality intended only for a restricted subset of callers. Such smart contracts require access control.
In this work, we develop an approach for modelling access control on the architecture level, using the Palladio tool. From this model, we automatically generate code stubs and source code which enforces access control. Furthermore, we generate a formal specification such that if the implementation adheres to this specification, access control is correct according to the model. We also describe our concrete experience with formal verification of the generated as well as the interactively written specification. Overall, our approach enables defining an access control model on a high level of abstraction, and ensuring its correct implementation.

1 Introduction

Ethereum smart contracts are programs, usually written in the Solidity programming language, which run on the Ethereum blockchain, managing digital currency and other assets on it. Numerous use cases have been suggested and implemented. However, safety and security of smart contracts has often been lacking, leading to a number of exploits causing great losses. Therefore, improving the security of smart contracts has become an active research area.

One major issue with Ethereum smart contract security has been the question of access control. In classical software development, most of a system is a "closed world": Execution is deterministic, and even for public functions, it is clear by whom (e.g., by which classes or components) and in which context they can be called. Only at the system boundary, e.g., where user interaction or external systems are involved, is access control required.

In contrast, the Ethereum smart contract network is an "open world": every public function of smart contracts can be accessed by every account on the blockchain. In practice, some parts of a contract's functionality must be inaccessible to all but a specific set of network participants (think of the "withdraw" functionality of a bank). If that is the case, e.g., if using some functionality or writing to a certain storage area must be limited to a certain set of accounts, some form of access control must be used. The design of the access control and the correctness of the implementation of this design are an important security property of Ethereum smart contracts.

In this work, we design an approach for achieving a correct implementation of a role-based access control. Our approach starts from modeling a smart contract application and its access control model on a high level of abstraction in the Palladio Component Model. From the model, we automatically generate parts of the application source code, as well as specification annotations for formal verification of parts that cannot be automatically generated. Finally, we describe how to formally verify the correctness of the implementation of the access control.

Outline Section 2 provides the necessary foundations for our approach. As a running example, we use an auction system on the blockchain. We describe it in Section 3.

In Section 4 we describe how model driven engineering approaches are used to generate code and formal specification for verification. For this purpose, the Palladio Component Model (PCM) of Reussner et al. [2] is applied to the domain of smart contracts in Section 4.1. A model is presented in Section 4.2 which extends the PCM to enable automatic code generation for Solidity smart contracts. The model contains the application's storage variables, its functions, and the access control model. The access control model contains the existing roles and a description which storage variables and which functions each role may access. With these models, a skeleton implementation of the application, consisting of function stubs and storage variables, is automatically generated. This approach is described in Section 4.3.

Correctness of the access control and its usage is then achieved as follows: The generator is extended to generate `modifiers`, which limit access to functions according to the role model. This extension is described in Section 4.4. Our approach to limiting access to storage is described in Section 4.5: We generate formal specification describing which parts of the storage each function may access. Formal verification tools can then be used to show that the implementation actually conforms to the generated formal specification. From the combination of limiting access to functions and limiting each function's access to storage, we achieve correct access control to storage.

Finally, we use formal methods to verify that the implementation of the access control is correct (Section 5.2), and that an attacker cannot assume a role in an unintended way (Section 5.3).

2 Foundations

In this section, the foundations for the approaches are provided. Generation of source code and model-driven engineering is described in Section 2.1. For the generation of the source code, the PCM is used as an architectural model (Section 2.2). The Solidity programming language is the de-facto standard for

Ethereum smart contracts. We give a short introduction in Section 2.3. The foundations of role-based access control are provided in Section 2.4.

2.1 Model-Driven Engineering

To cope with the complexity of software systems, models can be used to describe systems of different levels of abstraction. We use the term *model* according to the definition of Stachowiak [4]. In this definition a model is a representation of a real- or virtual world entity, created for a certain purpose. The representation of the entity only contains attributes for that purpose. As stated by Schmidt [3], in model-driven engineering (MDE), domain specific languages describe a system capturing different aspects like the application structure or behavior. Models can be used as input for approaches to analyse, for example, quality characteristics like performance, reliability or security. The Palladio approach of Reussner et al. [2], for example, uses models to represent different views on the system and provides an analysis approach to predict the performance of a system. Another aspect mentioned by Schmidt [3] is the application of transformation engines and generators to synthesize different artifacts like source code or other models. When generating source-code from other models, e.g. from architectural models, correspondences can be established and used to link elements of different views of the system. These correspondences can be used for consistency preservation between the elements of the views or, as stated by Schmidt [3], between analysis information. As shown by Yurchenko et al. [5] MDE approaches can also support the verification and analysis of security capabilities by generating source-code and formal specification from enriched architectural models. These formal specification have then to be verified by a verification tool.

2.2 The Palladio Component Model

The PCM of Reussner et al. [2] is an architecture design language to specify the architecture of component-based software systems. The model can be used, for instance, to provide performance, reliability or security predictions in the design time. For this purpose, the PCM provides modelling capabilities for several views on component-based software architecture. The views include, for instance, structural, deployment and behavioral aspects in the context of component-based software systems. In this report, we focus only on the structural views.

The structural aspects of a system can be described in the PCM by the the `Repository` model and `System` model. In the `Repository` model, components and interfaces are first class entities. `BasicComponents` are the basic building blocks of the software system and are used to describe a single blackbox component. `CompositeComponents` are used to assemble components out of other components. In the PCM, an `Interface` groups functionalities which are represented as `OperationSignatures`. Each `OperationSignature` contains `Parameters` and a `ReturnType`. The repository also provides predefined `PrimitiveDataTypes`, for example *string* or *bool*. For defining more complex types, the nameable

CompositeDataTypes can be used, which is composed of other DataType elements. To assign if a component provides or requires functionality in an interface, the OperationProvidedRole or OperationRequiredRole elements exist, which connect a component and an interface. The System model can be used to describe which components are connected and interact with each other in the software system. For this purpose components are instantiated in a System element by creating an AssemblyContext that references a component in the Repository model. The AssemblyConnector element connects the OperationProvidedRole and an OperationRequiredRole of the same interface of two components to describe that a component uses the functionality provided by another component.

2.3 Solidity

On the Ethereum blockchain, smart contracts exist in EVM bytecode. Bytecode is usually not directly written. Rather, smart contracts are written in Solidity, a high-level programming language, and then compiled to EVM Bytecode. Since our approach targets smart contract development, Solidity is our target language.

Solidity is "contract-oriented" [1] in that it provides some primitives which are useful primarily for smart contracts, such as cryptographic primitives for signing and encrypting, a built-in address data type, and keywords for identifying the caller of a function or the amount of currency transferred in a function call.

A smart contract consists of a set of storage variables pointing to (sets of) locations in the storage of the EVM, and a set of functions. Functions with the `private` modifier can only be called from within a contract, while public functions can be called by any account (i.e., any user or any other smart contract).

Additional function modifiers can be defined by the developer. Modifiers are reusable pieces of code that execute statements before or after functions consisting of a modifier head and a modifier body. The code to be executed is contained in the modifier body. The statement `_;` marks where the code of the modified function is executed. The usage of a modifier in a function is specified in the function head by providing the modifier name.

`require` clauses are designed to check at runtime whether a condition is met. If it is not, the function reverts, i.e., its effects on the global state are not applied, but no exception is thrown. In contrast, the `assert` statement throws an exception if its condition is violated. `require` clauses are used for input sanitization, e.g., checking whether the call parameters are valid. This includes access control.

2.4 Role-based Access Control for Solidity

In our understanding, access control for smart contracts encompasses two things: Access to functions, and access to storage. Both have to be defined for each role in an application.

[1] https://github.com/ethereum/solidity

A role-based access control (RBAC) model specifies a set of roles $\mathbb{R}$. For the set $\mathbb{F}$ of functions of a Solidity smart contract and the set $\mathbb{S}$ of storage variables of the contract, the RBAC specifies two relations $R_f : \mathbb{R} \times \mathbb{F}$ and $R_s : \mathbb{R} \times \mathbb{S}$, specifying which role has access to which function calls or storage variables, respectively.

R_f The relation R_f maps roles to functions. $(r, f) \in R_f$ means that someone who has role r has access to function f. $(r, f) \notin R_f$ means that when r calls f, f should do nothing, except possibly reporting an error.

R_s R_s maps roles to storage. $(r, s) \in R_s$ means that someone who has role r is allowed to access storage variable s. Since storage can only be changed through function calls, this can be decomposed into the requirement that a function which modifies s must limit access to those roles r for which $(r, s) \in R_s$.

In general, the smart contract application must enforce the following condition:

$$\forall f \in \mathbb{F}, s \in \mathbb{S}, r \in \mathbb{R} : \mathit{modifies}(f, s) => ((r, f) \in R_{\mathrm{f}} => (r, s) \in R_{\mathrm{s}})$$

2.5 Solc-verify

SOLC-VERIFY [1] 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.

3 The Auction Case Study

As a running example, an auction case study is introduced, where users can sell items on the blockchain. Sellers can create an auction from a central *AuctionManager* smart contract where the function *createNewAuction* receives the item to be sold and the expiration time of the auction. The *AuctionManager* contract creates a new *SingleAuction* contract for the auctioning of a single item.

The *SingleAuction* contract contains the three storage variables *managerAddress*, *sellerAddress* and *auctionClosed*. The *managerAddress* contains the address of the *AuctionManager* smart contract, while the *sellerAddress* contains the address of the seller. Both are set in the constructor and cannot not be changed afterwards. The *auctionClosed* is of boolean type and contains the information whether the auction is still active.

Bidders can bid a certain amount of money by calling the *bid* function. The function is `payable`, so that the amount they want to bid can be transferred with the function call. Sellers and auction managers should not be able to make bids for their own auctions. Therefore, they are not allowed to call the *bid* function. Bidders can withdraw their funds if they are currently not the highest bidder. After the expiration time, either the seller or any bidder can close the auction (setting `auctionClosed` to `true`) by calling the *close* function. After the auction is closed, the seller can withdraw their money by calling *sellerWithdraw*.

Each *SingleAuction* smart contract provides the function *emergencyShutdown* that can be called by an auction manager which aborts the auction (destroys the smart contract) and returns all funds transferred so far to the bidders.

An access control smart contract *AccessCtrl* is created for each *SingleAuction* contract to restrict the access to the functions to avoid misuse of public functionality (e.g. to avoid that the seller can cancel the auction by calling *emergencyShutdown* when a certain sum is not reached). For our auction example, the roles are determined when the *SingleAuction* contract is created. Therefore, we chose a static access control where the participants of the auction are defined in advance and cannot be modified afterwards.

The *AccessCtrl* contract can be used to check if an identity has a certain role with *checkId*. In the access control, the seller is identified by the role *Seller* and the *AuctionManager* smart contract by the role *Manager*. Bidders are identified by the role *Bidder*. Table 1 shows which roles may access the provided functions of the *SingleAuction* smart contract. Also Table 2 shows which roles may modify the storage variable of the *SingleAuction* smart contract.

Function	**Roles**
bid	Bidder
close	Bidder, Seller
emergencyShutdown	Manager
withdraw	Bidder
sellerWithdraw	Seller

Table 1: Mapping of functions to roles which may access them

Storage Variables	Roles
auctionClosed	Bidder, Seller
managerAddress	-
sellerAddress	-

Table 2: Mapping of storage variables to roles which may modify them

4 Generation of Code for Correct Access Control

The Palladio Component Model (PCM) of Reussner et al. [2] is already applied in the development, analysis and verification of component-based software systems. Yurchenko et al. [5], for instance, applies the PCM and MDE to support system correctness by generation of proof obligations for verification of information flow correctness into the source code.

To support software engineering in smart contract architecture development and verification with existing models and tools, we explore how the PCM can be applied for ensuring access control correctness in the domain of smart contracts. For this purpose, the applicability of the PCM to the domain of smart contracts is explored in Section 4.1 by modelling the case study described in Section 3. In Section 4.2, we extend the PCM to research the application of MDE and the PCM in the context of smart contracts to ensure access control correctness. The extended model enables the definition of access restrictions by assigning roles to storage variables and functions. In Section 4.3, we develop a generator for code stubs which builds on the extended model. Furthermore, we generate code enforcing access restrictions (Section 4.4) and specification annotations which ensure (if verified) that access control to storage variables is handled correctly (Section 4.5). Finally, we discuss the applicability of the PCM in the domain of smart contracts in Section 4.6.

4.1 Modelling Smart Contracts in Palladio

We develop a mapping from smart contracts to elements in Palladio's `Repository` model and `System` model, in order to explore the applicability of the PCM to model smart contract architectures. Figure 1 shows the `Repository` model that results from applying this mapping to our case study.

In general, components consist of more than one entity. In the context of smart contracts, however, we assign one contract to a `BasicComponent`. Therefore, the terms "smart contract" and "component" are used interchangeably in the context of the PCM.

To group strongly related public functions in smart contracts, `Interface` elements are used. The functions can be represented as `OperationSignatures` with their `Parameters` and `Return` type. The functionality for access control enforcement in the running example is described in the `Interface` *AccessControlling*, which contains the functionality *checkId* to check whether a certain role is assigned to the given identity.

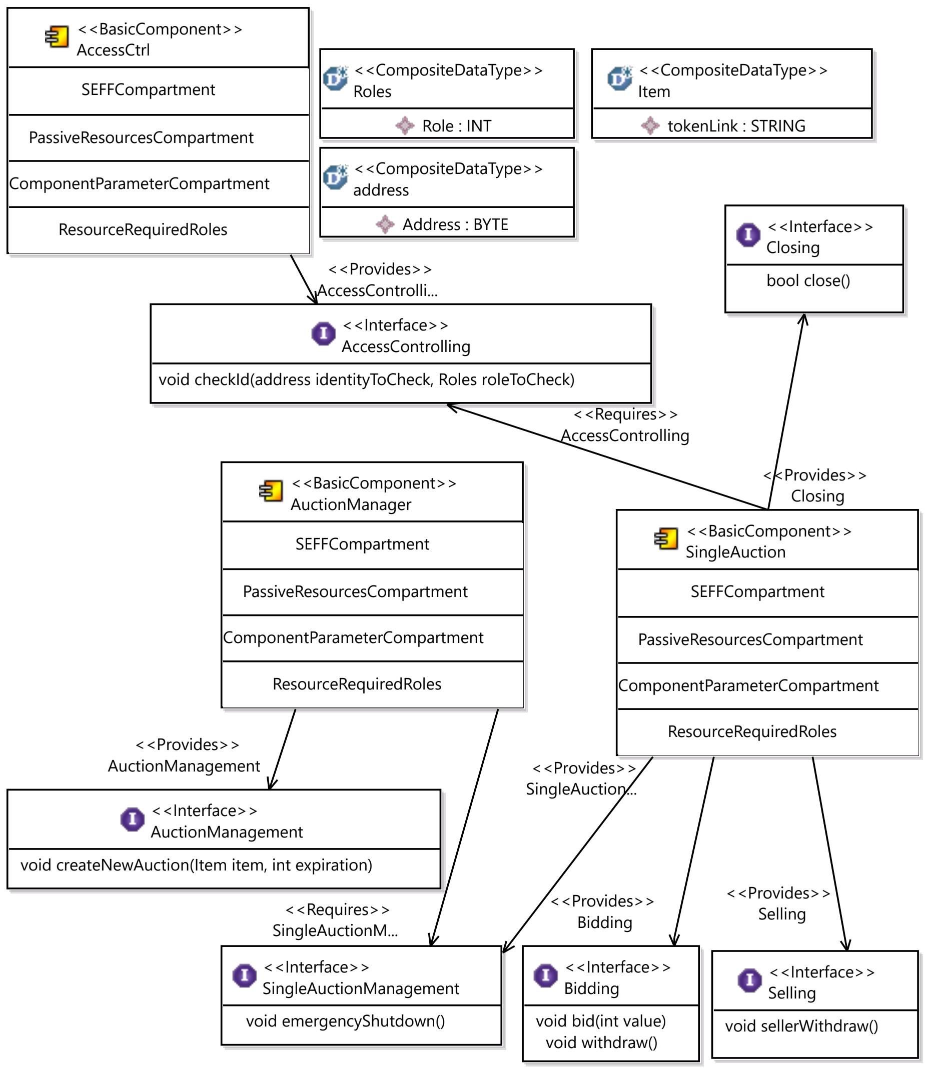

Fig. 1: Repository model of the auction case study

The PCM `PrimitiveDataType` *String*, *Boolean*, *Integer* and *Byte* can be mapped to their Solidity equivalents. In Solidity, the `address` type is predefined for unique identification of users and smart contracts. In the PCM, no such `PrimitiveDataType` exists. Therefore, a corresponding `CompositeDataType` has to be defined manually (see Figure 1).

Further Solidity data types which cannot be natively mapped to PCM data types (e.g., enums) are mapped by using a `CompositeDataType`.

When a smart contract declares certain functions as public (encapsulated in a `Interface`), an `OperationProvidedRole` role can be specified. For instance, the smart contract *AccessCtrl* in Figure 1 provides the functionality of *Access-Controlling*, e.g., checking whether an identity has a certain role. Therefore, a `OperationProvidedRole` connects the `BasicComponent` *AccessCtrl* and the `Interface` *AccessControlling*.

Similarly, when a smart contract requires functionality of another contract, an `OperationProvidedRole` is specified. The smart contract *SingleAuction* needs functionality of the *AccessControlling* interface (e.g., to check whether the caller of the *emergencyShutdown* functionality has the role of *Auction Manager*). Therefore, a `OperationRequiredRole` is created for the component of *SingleAuction* and the *AccessControlling* interface.

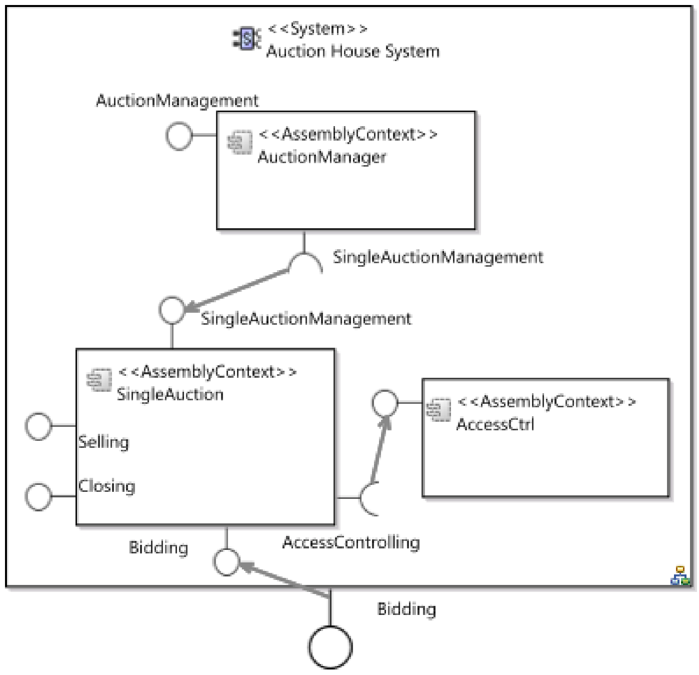

Fig. 2: System model of the auction case study

To describe relation between the smart contracts, the `System` model is used. Figure 2 shows the system model for the auction case study. Each smart contract is represented as a single `AssemblyContext`. Since, in the PCM, there is no information about the runtime instantiation of the contracts, only the static relation between their definitions can be described.

To describe which smart contract is called by another smart contract for using a certain functionality, an `AssemblyConnector` is created between an `OperationProvidedRole` and an `OperationRequiredRole` of the `AssemblyContexts` of both smart contracts. In Figure 2, the *SingleAuction* smart contract uses the functionality of the *AccessCtrl* smart contract. Therefore, an `AssemblyConnector` is established between the the provided and required roles of *AccessControlling* from the `AssemblyContexts` of *AccessCtrl* and *SingleAuction*.

4.2 Extending the Palladio Component Model for Role-based Access Control Information

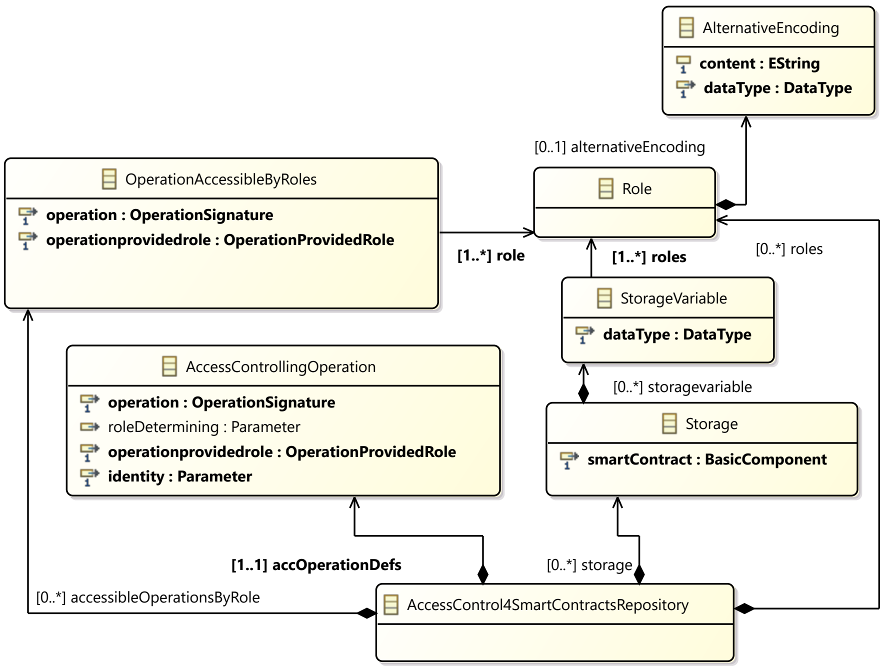
AlternativeEncoding
content : EString
dataType : DataType
[0..1] alternativeEncoding
OperationAccessibleByRoles
operation : OperationSignature
operationprovidedrole : OperationProvidedRole
Role
[1..*] role
[1..*] roles
[0..*] roles
StorageVariable
dataType : DataType
AccessControllingOperation
operation : OperationSignature
roleDetermining : Parameter
operationprovidedrole : OperationProvidedRole
identity : Parameter
[0..*] storagevariable
Storage
smartContract : BasicComponent
[1..1] accOperationDefs
[0..*] storage
[0..*] accessibleOperationsByRole
AccessControl4SmartContractsRepository

Fig. 3: Model `rbac4smartcontracts` for the PCM

The model we use to extend the PCM for our approach is called `rbac4smartcontracts` [2]. The elements of the `rbac4smartcontracts` model are shown in Figure 3.

The `Role` element contains a `name` and is used as a label representing a role. Therefore, three `Role` elements with the names *Manager*, *Seller* and *Bidder* exist for the auction case study. In general, it is assumed that `Roles` are identifiable by their name. However, the `AlternativeEncoding` enables alternative descriptions for `Roles` by providing a data type and an arbitrary value.

The `OperationAccessibleByRoles` element describes the Relation R_f (see Section 2.4), i.e., that the `OperationSignature` *operation* in the `Interface` provided by the `BasicComponent` (defined through the `OperationProvidedRole`) may only be accessed by the specified roles. For example, *close* in the interface *Closing* provided by the smart contract *SingleAuction* may only accessed by the roles *Seller* and *Bidder*.

To describe the mapping R_s (see Section 2.4), it is necessary to assign roles to storage variables of smart contracts. No storage variables can be specified in the standard PCM, because components are modelled as blackboxes, with no internal information exposed. However, for the purposes of this work, we relax the blackbox principle, and introduce the `Storage` model element to the `rbac4smartcontracts` model, as a container for storage variables. Within it, the `StorageVariable` element represents a storage variable, consisting of a name, a `DataType`, and a set of references to `Roles` that are allowed to modify the variable. To assign all variables in the `Storage` to a smart contract, a `BasicComponent` is referenced.

The element `AccessControllingOperation` defines which operation provided by a component is used to check whether a identity has a certain role. An `OperationSignature` and an `OperationProvidedRole` is referenced for this purpose (similar to `OperationAccessibleByRoles`).

roleDetermining references a `Parameter` element to specify that this parameter in the specified `OperationSignature` is used as input for the role to check for. This parameter is optional if there is only a single role in the system, and if the access control only checks whether the identity has this role assigned or not. For the access checking operation, it is assumed that the operation contains at least one parameter providing the identity to be checked. Therefore, the reference *identity* marks the `Parameter` in the `OperationSignature` that is used to input the identity to be checked.

In Ethereum smart contracts, the address of an account is used for identification. In the PCM model of the auction example, the *AccessCtrl* smart contract provides the function *checkId* to check whether an identity has a certain role. Therefore, the `AccessControllingOperation` indicates the `OperationProvidedRole` of the `BasicComponent` *AccessCtrl* to the `Interface` *AccessControlling* and also references the `OperationSignature` *checkId*. Furthermore, *identity* defines the parameter *identityToCheck* to be used as input of

[2] https://github.com/KASTEL-SCBS/PCM2Solidity

```
pragma solidity ^0.5.0;

import "./AccessCtrl.sol"; //TODO: Modify Import, if
    structure is changed!

contract SingleAuction {

  AccessCtrl ac; //TODO: Auto-generated Field!

  bool auctionClosed; //TODO: Auto-generated Field!
  address managerAddress; //TODO: Auto-generated Field!

  function emergencyShutdown() public{
    // TODO: implement and verify auto-generated method stub
    revert("TODO:␣auto-generated␣method␣stub");
  }

  function close() public returns (bool output){
    // TODO: implement and verify auto-generated method stub
    revert("TODO:␣auto-generated␣method␣stub");
  }
}
```

Listing 1.1: Generated code stubs for the *SingleAuction* smart contract

the identity to be checked for the role provided by the parameter *roleToCheck* defined by the *roleDetermining* reference.

4.3 Generation of Solidity Code Stubs

We now build on the extended PCM to automatically generate code stubs for Solidity smart contracts. For this, we develop a model-to-text generator [3]. It takes the `Repository` model and `System` model as well as the extension model described in Section 4.2 and generates smart contract stubs. Listing 1.1 shows a generated stub of the *SingleAuction* smart contract.

For each defined `BasicComponent`, a Solidity file containing the smart contract definition with the component's name is generated. For each smart contract, the provided functions are generated directly from the `OperationSignatures` of the `Interface` in an `OperationProvidedRole`.

In Listing 1.1, the generated functions *emergencyClose* of the `OperationProvidedRole` to the interface *SingleAuctionManagement* as well the function *close* from `OperationProvidedRole` to the interface *Closing* is shown. In structural blackbox component-based architectural descriptions, no information about the implementation of a functionality available. Therefore, the bodies of the generated Solidity functions contain only a `revert` statement.

[3] also found under `https://github.com/KASTEL-SCBS/PCM2Solidity`

```
pragma solidity ^0.5.0;


contract AccessCtrl {
  enum Roles { Manager, Bidder, Seller }

  function checkId(address identityToCheck, Roles roleToCheck
      ) public  {
    // TODO: implement and verify auto-generated method stub
    revert("TODO:␣auto-generated␣method␣stub");
  }
}
```

Listing 1.2: The generated `Roles` enum

Smart contracts can call the functionality of another smart contract on an instance of that contract's type. Therefore, when an `AssemblyConnector` between two `AssemblyContext` exists, a field with the type of the providing contract is generated in the requiring smart contract. Furthermore, an `import` statement for the providing smart contract is generated. In the auction system model, as shown in Figure 2, the *SingleAuction* contract calls functions from the *AccessCtrl* contract. Therefore, a field with the type of *AccessCtrl* is generated.

The *enum* type in Solidity provides a convenient way of expressing which roles exist in an application. However, in the PCM, *enum* data types cannot be specified. We solve this problem by creating a `CompositeDataType` called `Roles` in the PCM. When the generator encounters this in the `roleDetermining` parameter of `AccessControllingOperation`, a role enum with every role in the `rbac4smartcontracts` model is generated in the component referenced through the `OperationProvidedRole` in `AccessControllingOperation`.

The example in Listing 1.2 shows the *AccessCtrl* smart contract with the *enum Roles*, consisting of the elements *Manager*, *Seller* and *Bidder*.

For each `StorageVariable` in a `Storage` container, a field with the given name and the type corresponding to the referenced `DataType` is generated into the referenced smart contract (e.g., *auctionClosed* in Listing 1.1).

4.4 Generation of Access Control Modifiers

Access restrictions to functions can be realized by generating code that forces authorization checks w.r.t. the role model before executing the function. Solidity provides `require` clauses (see Section 2.3), which can be used for this purpose. For reusability and readability, the `require` clauses can be encapsulated in a modifier (see ibid.). To enforce access control, we generate modifiers which call an access control functionality in a require clause to check if the caller of the function has the correct access rights. Listing 1.3 shows the modifiers generated for our running example.

```
pragma solidity ^0.5.0;

import "./AccessCtrl.sol"; //TODO: Auto-generated Import!

contract SingleAuction {
  AccessCtrl ac; //TODO: Auto-generated Field!

  bool auctionClosed; //TODO: Auto-generated Field!
  address managerAddress; //TODO: Auto-generated Field!

  function emergencyShutdown() public onlyManager {
    // TODO: implement and verify auto-generated method stub
    revert("TODO:␣auto-generated␣method␣stub");
  }

  function close() public onlyBidderSeller returns (bool
      output){
    // TODO: implement and verify auto-generated method stub
    revert("TODO:␣auto-generated␣method␣stub");
  }

  modifier onlyBidderSeller {
    require(ac.checkId(msg.sender, AccessCtrl.Roles.Seller)
      || ac.checkId(msg.sender, AccessCtrl.Roles.Bidder),
       "Only␣acessible␣by␣role(s):␣Seller,␣Bidder.");
    _;
  }
  modifier onlyManager {
    require(ac.checkId(msg.sender, AccessCtrl.Roles.Manager),
      "Only␣acessible␣by␣role(s):␣Manager.");
    _;
  }

}
```
Listing 1.3: *SingleAuction* with generated modifiers

For a role combination in the `OperationAccessibleByRoles` elements with `OperationProvidedRole`s for a specific component, a modifier is created in the corresponding smart contract. Let $\mathrm{R_m}$ be that combination for a given modifier. The name of the modifier is created from the ordered set of role names in $\mathrm{R_m}$ with the prefix *only*. If the same role combination is specified for different `OperationSignature`s for the same smart contract, only a single modifier is created.

In our running example, only the roles *Bidder* and *Seller* may use the *close* function in the *SingleAuction* smart contract. Therefore, the modifier *onlyBidderSeller* is generated in the *SingleAuction* smart contract.

The generator uses the information given by the `AccessControllingOperation` element to generate the modifier bodies. Every modifier body consists of a `require` statement. In this statement, the access control function (e.g. in the example *checkId*) of the smart contract, defined in the `AccessControllingOperation`, is called for all roles in $\mathrm{R_m}$. For each call to the operation in the `AccessControllingOperation`, the identity and the role to check are provided to the specified parameters *identity* and *roleDetermining*.

In Solidity, the identity of the caller of a function is their address, accessible via the `msg.sender` keyword. Therefore, `msg.sender` is used as input for *identity*.

When the data type of *roleDetermining* is *string* and no `AlternativeEncoding` is specified, the name of the role is converted to a string. If an `AlternativeEncoding` is specified, and the data type fits to the one of the *roleDetermining* parameter, the value in `content` is supplied. When using the generation of the *enum Roles* as described in Section 4.3 the *enum* value with the reference to the access control is generated as input for the *roleDetermining* parameter. This approach is shown in Listing 1.3. When none of these approaches fit, a `revert` statement is generated.

It is assumed that the access controlling operation only returns a boolean value as signal whether the role is assigned to the identity or not. If the return type of the `OperationSignature` in `AccessControllingOperation` is not of the `Bool` data type, no assertion to the require-statement can be made. Therefore, in this case, a `revert` statement is generated.

A generated `require` statement can be seen in the *onlyManager* modifier in the example in Listing 1.3. In the `require` statement, the function *checkId* of the *AccessCtrl* is called.

If more than one role may access a function, the checks are combined by an *or* relation (see the *onlyBidderSeller* modifier in Listing 1.3). We append the statement `_;` at the end of the modifier body, so that the code of the modified function is executed after the access control checks.

Finally, each modifier's name is added to the function header corresponding to the `OperationSignature` that is targeted by the `AccessibleOperationByRoles` element. In the running example, Listing 1.3 shows *onlyManager* in the *emergencyShutdown* function's header.

The specification on the architectural level makes the role model visible on a higher level of abstraction. With our generation approach, the confirmation if an identity may access the functionality can be deferred to a auto-generated, reusable set of modifiers. Through the automated generation, the role model is automatically enforced.

4.5 Generation of Formal Specification

Apart from ensuring that functions can only be accessed by authorized identities, we also want to guarantee that access to a smart contract's storage is properly limited. Access to functions can be limited in a generic way using constructs of the Solidity language. In contrast, correct authorization of storage access cannot be enforced through code generation alone. Instead, our approach is to generate formal specification such that if the application conforms to the specification, it also enforces correct access to storage.

To prove that the implementation conforms to its specification, we will use the SOLC-VERIFY tool. Therefore, the annotations will be generated in its specification language.

Let $\mathbb{F}$ be the set of functions of the smart contract, $\mathbb{S}$ the set of storage variables, and $\mathbb{R}$ the set of roles. Recall the relationship of the RBAC model and a function's modifiable storage in Section 2.4. This relationship can be rephrased as:

$$\forall f \in \mathbb{F}, s \in \mathbb{S}, r \in \mathbb{R} : ((r, f) \in R_f \wedge \neg(r, s) \in R_s) => \neg modifies(f, s) \quad (1)$$

We want to generate annotations expressing this relationship. Therefore, if role r may call function f, but may not modify storage variable s, then we must generate an annotation expressing that f may not modify s. We find the relevant pairs (f, s) by iterating over the corresponding model elements, i.e., `OperationAccessibleByRoles` and `StorageVariable`.

Strings are currently not comparable in Solidity, and therefore we cannot express String equality in a postcondition. Mappings and structs are not representable in the PCM without further extensions. Therefore, we only generate annotations for storage variables of the *int*, *bool*, and *address* types, as well as arrays of these types.

In SOLC-VERIFY a postcondition `c` is written above a function's header as a comment of the form `///@notice postcondition c`. To express that a function f may not modify a storage variable v, we state that v's value after the execution of f must be the same as before the execution.

In SOLC-VERIFY, the pre-state of a function can be accessed via the `old` keyword. Let the state of a storage variable s of type *int*, *bool*, or *address* after the execution of a function f be denoted as $\mathrm{S}_{\mathrm{s_f}}$ and $\mathrm{S}_{\mathrm{old(s_f)}}$ the state before the execution. Then the condition that s must not be modified can be specified as $\mathtt{S_{v_f}\ =\ S_{old(v_f)}}$. An example for the generated condition is shown in Listing 1.4.

For arrays, the generated specification must quantify over the array in order to express that all elements remain unchanged (see Listing 1.5)

```
pragma solidity ^0.5.0;

import "./AccessCtrl.sol"; //TODO:  Auto-generated Import!

contract SingleAuction {

  AccessCtrl ac; //TODO: Auto-generated Field!

  //Modifiable by: Bidder, Seller
  bool auctionClosed; //TODO: Auto-generated Field!
  //Modifiable by: Nothing
  address managerAddress; //TODO: Auto-generated Field!

  /// @notice postcondition auctionClosed ==
      __verifier_old_bool(auctionClosed)
  /// @notice postcondition managerAddress ==
      __verifier_old_address(managerAddress)
  function emergencyShutdown() public onlyManager {
    // TODO: implement and verify auto-generated method stub
    revert("TODO:␣auto-generated␣method␣stub");
  }

  /// @notice postcondition managerAddress ==
      __verifier_old_address(managerAddress)
  function close() public onlyBidderSeller returns (bool
      output){
    // TODO: implement and verify auto-generated method stub
    revert("TODO:␣auto-generated␣method␣stub");
  }

  modifier onlyBidderSeller {
    require(ac.checkId(msg.sender, "Seller") || ac.checkId(
        msg.sender, "Bidder"),"Only␣acessible␣by␣role(s):␣
        Seller,␣Bidder.");
    _;
  }
  modifier onlyManager {
    require(ac.checkId(msg.sender, "Manager"),"Only␣acessible
        ␣by␣role(s):␣Manager.");
    _;
  }
}
```

Listing 1.4: *SingleAuction* with generated formal specification

```
/// @notice postcondition forall (int i)
///     (!(0 <= i && i < a.length)
///  || a[i] == __verifier_old_int(a[i]))
  function f() public {
  ...
  }
```

Listing 1.5: Generated annotation stating that all elements of an array a must stay unchanged by function f

In the auction case study, the variable *auctionClosed* may only be modified by the roles *Bidder* and *Seller*. The function *emergencyClose* may only be accessed by the role *Manager*. Therefore, the function must not modify the variable *auctionClosed*. Likewise, an annotation is created for *emergencyShutdown*, stating that *auctionClosed* has to be the same before and after the execution. The variable *managerAddress* should not be modified after the constructor; an annotation expressing this is generated for every function.

4.6 Discussion: Applicability of the Palladio Component Model

The PCM was created for classical component-based software architectures. Because smart contracts differ from this view, the applicability of the PCM for smart contracts has to be discussed. As presented in Section 4.1 and Section 4.2, it is possible to use the PCM for describing smart contract architectures. However, there is a mismatch between components and smart contracts: While components generally consist of multiple entities (e.g. several classes), smart contract design is currently limited to a single entity. Therefore, the semantics of PCM components have to be adapted for our approach. The same is true for interfaces, whose role in Ethereum is different from the one in classical software systems. Furthermore, in Ethereum, there are data types like `address` or mappings which are not natively provided by the PCM.

Some problems can be mitigated in the generation of code, e.g., ignoring the interfaces for function stub generation, or using `CompositeDataType`s for data types not present in the PCM. The Necessity of describing the state of a smart contract, however, required a deviation from the blackbox principle of the PCM.

We conclude that for further research in the direction of this work, either the PCM has to be modified to be more fit for smart contracts, or other modelling tools must be used. The creation of a language explicitly tailored to describing access control aspects of smart contracts would be another alternative.

Nevertheless, for prediction of quality aspects in conjunction with architectures consisting of classical software components and smart contracts, it could be beneficial to introduce mechanisms for the alignment of smart contracts as well as classical software systems in the PCM. This approach could be used to extend existing analyses or creating new ones to cope with the challenge of analysing such architectures.

5 Proving Correctness of the Access Control

From the Palladio Component Model, we generated function stubs, Solidity modifiers, and formal specification concerning access control, as well as a code skeleton of the `AccessCtrl` smart contract.

However, neither the implementation of access control nor of the application itself can be automatically generated, and the correctness of the access control mechanism depends on both implementations. In order to verify the correct functioning of the access control, we have to prove that (1) the implementation of the functions conforms to the generated specification, (2) the access control itself works correctly, e.g., the `checkId(addr, owner)` function returns true only if the supplied address actually has the owner role, and (3) the roles can only be assumed in the intended way. In this section, we give an example access control implementation for the auction case study, and sketch how the above properties can be verified.

5.1 Verification of Generated Annotations

In Section 4.5, we generated annotations expressing that a function cannot change the state of a storage variable. The annotations are in the language of the SOLC-VERIFY tool, which we use for verification.

In our example, verification is almost trivial: Since the specification has already been generated automatically, we simply call the tool after implementation is finished. Verification is automatic and takes less than 10 seconds in our case (of course, this depends on the implementations's complexity). If SOLC-VERIFY reports any errors, the implementation has to be adapted accordingly.

5.2 Correctness of the Access Control Implementation

The automatically generated `AccessCtrl` smart contract provides the function `checkId(addr, role)` which is supposed to return `true` iff the address `addr` has the role `role`. We need to prove that this function behaves correctly.

Listing 1.6 shows an example access control for a fictional auction application. In order to evaluate SOLC-VERIFY's performance, we increase the complexity compared to our running example. Again, there are three roles: A manager, identified by a single address; a set of sellers, listed in the corresponding mapping; and bidders, who are listed in an array.

The specification in our example is a method contract for the `checkId()` function, written in the annotation for solc-verify, a tool for functional verification of Solidity smart contracts. The language allows first order logic and Solidity syntax. In our example, the specification states whether the function should return `true` for each of the three roles: For the Manager role, it should return `true` if the correct address is given; for the Seller role, it should return `true` if the corresponding mapping entry is `true`; and for the Bidder role, it should return `true` if the address is contained in the corresponding array.

```
pragma solidity ^0.5.0;
contract AccessControl {
    enum Roles {Manager, Seller, Bidder}
    address manager_addr;
    mapping(address => bool) sellers;
    address[] bidders;

/// @notice postcondition !( role == Roles.Manager )
///    || (res == (addr == manager_addr))
/// @notice postcondition !( role == Roles.Seller )
///    || (res == (sellers[addr]))
/// @notice postcondition exists (uint i) (!(role == Roles.Bidder)
///    || ((0 <= i && i < bidders.length && bidders[i] == addr) == res))
function checkIdentity(address addr, Roles role)
        public view returns (bool res) {
    if (role == Roles.Manager) {
        return addr == manager_addr;
    } else if (role == Roles.Seller) {
        return sellers[addr];
    } else if (role == Roles.Bidder) {
        bool ret = false;
        /// @notice invariant role == Roles.Bidder
        /// @notice invariant exists (uint u)
        ///         ((0 <= u && u < i && bidders[u] == addr) == ret)
        for (uint i = 0; i < bidders.length; i += 1) {
            if (bidders[i] == addr) {ret = true;}
        }
        return ret;
    } else { return false; }
    }
}
```

Listing 1.6: An example access control with formal specification

We now want to prove that our implementation conforms to the specification, using the SOLC-VERIFY tool. For the *Manager* and *Seller* roles, this works automatically; however, for the *Bidder* role, we do not succeed. The specification requires existential quantification in a disjunction, but even though we supply a sufficient loop invariant, SOLC-VERIFY fails to verify the correctness of the implementation.

We conclude that while our approach has merit, the performance of verifcation tools can be a bottleneck, depending on the complexity and the data structures used in the access control implementation.

5.3 Correctness of Role Management

Apart from the correctness of the access control's implementation, there are further points to be considered in order to ensure overall correctness of the access control mechanism: In particular, even if the `checkId()` function works correctly, it must be ensured that roles can only be assumed in the correct, intended way, e.g., it must be impossible for an attacker to insert his address into the `sellers` mapping. Furthermore, there may be additional restrictions in the Palladio model which need to be translated into formal specification.

There is no generic way to specify or even verify the overall correctness of role management, but we will consider some examples.

Mutual exclusion Consider the extended auction example from the previous section. In the model, it is specified that the Bidder role is mutually exclusive with the other roles, i.e., if `checkId(addr, Bidder)` is true, `checkId(addr, Seller)` and `checkId(addr, Manager)` must both return false. This entails that if an address is contained in the `sellers` mapping, it must not be contained in the `bidders` array. This, in turn, is an invariant of the auction smart contract, which every function must maintain. It can be formalized as follows (with $\mathbb{A}$ the set of addresses):

$$\forall a \in \mathbb{A} : \neg checkId(a, bidder) \vee \neg checkId(a, seller)$$

Such an invariant could be automatically generated from the archtitecture model. However, this invariant contains function calls, which are usually challenging in formal verification. Some tools, like SOLC-VERIFY, do not allow function calls in the annotation language. In order to avoid function calls, the property can be expressed on the implementation level:

$$\forall a \in \mathbb{A} : \neg(sellers[a] \wedge bidders.contains(a))$$

While this formalisation makes verification easier (or at all feasible), it is implementation specific and cannot be automatically generated.

Dynamic Role Management If the roles are static, i.e., role assignements do not change at runtime, it is enough to verify that the access control is implemented correctly. However, if roles can be granted or taken away at runtime, this process also has to be considered for correctness.

As an example, consider the above auction example, but with an added functionality which enables the contract's manager to transfer the manager role to a different address, and which also enables the manager to add further addresses to the set of sellers, or remove addresses from it.

In this scenario, correctness of role management means that *only* the current manager address can change the manager address, and that *only* the manager address can change the mapping which stores the seller addresses. Therefore, the question of access control is connected to the question of which functions can modify the respective storage locations.

A practical approach in our auction example would be to add the `onlyManager` modifier to all functions which can change the manager address. For all other functions, the solc-verify tool can be used to prove that they can in fact not modify the manager address by way of a `modifies` annotation. Listing 1.7 shows an example for such an annotation, stating that the function must only modify the `balances` mapping at the `msg.sender` key. A proof that the function conforms to its specification therefore implies that the function cannot access either the `manager` address nor the `sellers` mapping.

```
/// @notice modifies balances[msg.sender]
function bid() public payable returns bool {...}
```

Listing 1.7: A `modifies` annotation

Another, more convenient approach would be whitelisting, i.e., annotating each state variable with a list of functions that are allowed to modify it, and then automatically generating the corresponding formal specification. However, there is currently no tool which enables this kind of whitelisting specification.

6 Conclusion and Outlook

In this report, we demonstrated the use of high-level modelling tools in combination with formal methods in order to guarantee a security property of Ethereum smart contracts, namely, correctness of access control. We adapted the Palladio tool so that it can handle the constructs that are necessary for this, and implemented automatic generation of Solidity source code, including function stubs, access modifiers, and specification annotations. Furthermore, we discussed how formal methods can be used to prove the correctness of the final implementation, showing a comprehensive approach to correct access control for smart contracts.

We noted some shortcomings of the Palladio tool in the context of smart contracts. In the future, we could either further extend Palladio, enabling the handling of all necessary Solidity constructs. Another possible way would be to implement a standalone tool, building on the experiences of this work.

References

1. Hajdu, Á., Jovanović, D.: solc-verify: A modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 161–179. Springer International Publishing, Cham (2020)
2. Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K.: Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press, Cambridge, MA (October 2016), http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures
3. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer 39(2), 25–31 (Feb 2006)
4. Stachowiak, H.: Allgemeine Modelltheorie. Springer, Wien, New York (1973)
5. Yurchenko, K., Behr, M., Klare, H., Kramer, M., Reussner, R.: Architecture-Driven Reduction of Specification Overhead for Verifying Confidentiality in Component-Based Software Systems. In: Proceedings of MODELS 2017 Satellite Event (MoDeVVa Workshop), co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017). vol. 2019, pp. 321–323. CEUR-WS (September 2017)