Static Capability-based Security for Smart Contracts

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—Smart contracts manage resources on a blockchain platform. These resources exist in the form of cryptocurrency, but also, more generally, in the form of data that is stored on the ledger. Due to the peculiarities of blockchain networks, changing smart contracts after deployment is hard or even impossible. This means that smart contracts must be correct and secure upon deployment. However, frequent exploits show that smart contract security is still difficult to achieve.**

To address this problem, we propose a static approach for capability-based smart contract security. We identify three central capabilities: calling functions, modifying state, and transferring currency. The entities to which these capabilities are attached are accounts (organized in roles) and smart contract functions.

In our approach, a developer, given a security policy for a smart contract application, first designs a model of the application. The model consists of state variables, functions, roles and capabilities. We provide a definition of when the created model is consistent, and develop a formal analysis of model consistency. Furthermore, we provide a definition of what constitutes a secure implementation w.r.t. this model, and describe how to achieve an implementation which fulfills this notion of security.

***Index Terms*—Smart contracts, Security, Model-driven development, Formal analysis**

I. Introduction

Smart contracts are programs which run in a blockchain network. They grant access to resources that are stored on the blockchain, such as cryptocurrencies or tokens representing real-world assets. Smart contracts are unique in that they allow anyone to run a program on a different computer, while still being certain about the execution semantics, and about what source code is being executed.

Another characteristic of smart contract applications is their reactive-system-like interface: They expose public functions which can be called by everyone in the network, including other functions. Smart contract applications live in an open world – as a matter of principle, any agent on the network can see and call any public function. Limiting access to resources, therefore, is a responsibility of the functions themselves. Furthermore, calling a function is the only way to transfer funds or change the state of an application.

The unique traits of smart contract applications come at the price of immutability: Once deployed, the source code smart contracts generally cannot be changed. Furthermore, the source code, including possible programming errors, is usually publically available. This means that any vulnerability has a large probability of being exploited, and since smart contracts cannot be patched, it is very important that smart contracts are secure upon deployment. However, a long and ongoing history of vulnerabilities and exploits of smart contract applications (cf. [**?**]) shows that security in this field is very much a pressing issue.

In this work, we take a view of smart contract security that is based on resources, and who can access them. We identify three main resources: the state of an application, its public functions, and cryptocurrency. Inspired by the influential capability-based security model [11], we define capabilities as access to resources (calling functions, changing state, and transferring currency) in the context of smart contracts.

Typically, capability-based security is implemented for operating systems, where capabilities are determined at run-time. We argue that in the open, highly adversarial environment of smart contract platforms, all relevant capabilities for an application should be determined at design time, based on the security policy given in the requirements specification.

Experience shows that building security mechanisms in source code is complex and error-prone. Directly implementing a security policy in a smart contract programming language poses a serious risk of exploits. This risk is exacerbated by the fact that smart contracts cannot easily be patched. Therefore, we argue that a suitable abstraction is needed, which helps a smart contract application developer focus on the relevant aspects. Thus, in this work, we develop a metamodel of smart contract applications, enriched with capabilities.

After designing an application by instantiating this metamodel, a developer must have a way to decide whether their model is *consistent* and *precise*. Therefore, we develop a formal definition of consistency and precision based on set-theoretic semantics of our metamodel. Then, we develop analysis techniques which enable a developer to detect whether their model contains contradictions, or whether the specified capabilities can be more restricitve.

Given a satisfactory model of an application, the next task is to develop an implementation that is correct w.r.t. the model. For this, we propose an approach for implementing an application for the Ethereum platform written in Solidity. Our approach is based on a combination of code generation and formal analysis. Since our metamodel is platform-independent, the approach can be used for other platforms in principle, but needs to be adapted accordingly.

In summary, our contribution is three-fold:

- A metamodel for smart contract applications with a capability-based security model
- A definition of consistency and precision of a given model, and analysis techniques for proving consistency and precision
- A partly automatic process for developing a platform-specific implementation which is secure in the sense of the model

Overall, we hope that this work enables a developer to turn the security policy of a requirements specification into an implementation which respects this security policy at all times.

The rest of this paper is organized as follows: Section II surveys the current state of research in the field of smart contract security. Section III gives an overview of the characteristics of smart contract platforms, and the implied consequences for designing secure applications. Section IV introduces our metamodel for smart contract applications and capabilities. Section V gives a formal semantics to the capability definition. Section VI gives a definition of what constitutes a consistent and precise model, and our approach to analyse these model properties. Section VII describes how to turn a model into a conformant implementation. Section VIII gives an overview of the process of developing a secure implementation from an initial security policy. Section IX concludes.

II. Related Work

Smart contract security has been quite an active research field over the last years, in part owning to a number of high-profile exploits. Much effort was invested in developing static analysis tools for smart contracts on the Ethereum platform. These target either EVM bytecode (e.g., Mythril [12] and Securify [16]) or source code in the most common high-level Ethereum programming language Solidity (e.g., [5]). By conducting symbolic execution or related analysis techniques, these tools can detect the presence of predefined vulnerabilities. They often work on an abstract representation of a programm, e.g., a call graph, which can be useful for the analysis of an implementation in the scope of this work (cf. Section VII).

Furthermore, several tools for formal verification of smart contracts have been developed. These allow specifying the intended behaviour of a programm in formal logic, and conducting a proof of correctness of an implementation against this specification. Examples for formal verification tools are SOLC-VERIFY [7] and Celestial [4], which allow specification of contract-level invariants and function contracts written in first-order logic. Other tools such as VerX [13] and SmartPulse [15] can be used to specify and verify temporal as well as safety and liveness properties. As with static analysis tools, we use formal verification as one method for achieving an implementation which conforms to a given model.

The 2vyper verification tool [2] works on smart contracts written in the Vyper programming language. Apart from invariants and history constraints, 2vyper also has a concept of resources and ownership: It distinguishes between having and owning a resource (e.g., cryptocurrency, or a token). This is related to the notion of transfer capabilities of this work.

As opposed to all the works above, we approach smart contract security from a modelling perspective, where the developer first designs an abstract representation of the application. Modeling smart contract applications is also an active research field. Early approaches envisioned to automatically generate smart contracts from natural language [6]. DasContract [14] is a visual language for modelling smart contract applications with code generation capabilities. There is also recent work designing model-driven architecture-based development for smart contracts [8]. While we draw inspiration from these works, we focus on one particular set of properties of smart contracts, and strives to design a process which ensures these properties in a rigorous way.

There are also works where smart contract applications are modelled in system specification languages. One such approach uses Event-B as a modelling language and verifies safety properties on the model before manually implementing it [19]. Another approach [18] specifies applications and models participant behaviors in TLA+ [10]. On this model, known security vulnerabilities can then be detected by the TLA model checker.

An approach which is comparable to ours is Quartz [9], where developers can design an application in a domain-specific modelling language. The model is then translated to TLA+, and any specification of the intended behaviour can be given to a model checker. When a satisfying model has been found, Quartz also enables code generation of correct-by-construction smart contracts. While the overall process is similar, our work focuses on security instead of safety, and our metamodel explicitly includes security-relevant aspects. Implementation-wise, the Quartz model also requires extensive specification of function behaviour. In our approach, the implementation of functions is done by the developer and checked for conformance afterwards, since we judged that otherwise the metamodel will either become very complex, or have to be overly restrictive.

III. Smart Contract Characteristics

For the purpose of modelling smart contract applications, we do not consider a specific platform. However, in this section, we will give a brief overview of our notion of smart contracts and what we consider to be their defining characteristics. These are reflected in our metamodel of smart contract applications and in our definition of capabilities.

Smart contracts, for our purposes, are characterized by:

a) An open world: Everyone can participate in smart contract platforms, in particular by calling publically exposed smart contract functions. Access to these functions cannot be limited except within the functions themselves. All security policies, therefore, must be implemented on the source code level.

b) Centralized view of state: Smart contract platforms create a unified view of state through some consensus mechanism. This state is public as a matter of principle. Therefore, we do not consider read access a relevant question in the context of smart contracts.

c) State change through functions: Calling a function is the only way to change the state. The overall state is partitioned into namespaces: Each contract defines its own namespace, which can only be modified by the contract's own functions.

d) Transactionality: Smart contract functions are transactional, i.e., they either terminate successfully, or revert without changing the state at all. The revert mechanism can be used to implement access control: A function call which reverts is guaranteed to leave the state unchanged, and can therefore, in the absence of a mechanism to prevent the call itself, be used to signal that access has been denied.

e) Built-in Currency: Most smart contract platforms define tokens, referred to as cryptocurrency, that can be transferred using built-in mechanisms. Cryptocurrency plays an essential role in smart contract applications and ecosystems.

IV. Modelling Applications and Capabilities

This section describes our metamodel of smart contract applications and roles, and a simple grammar for expressing capabilities.

A. Smart Contract Applications

We developed a metamodel of smart contract applications that developers can instantiate with a model of an actual application. The metamodel is shown in Figure 1.

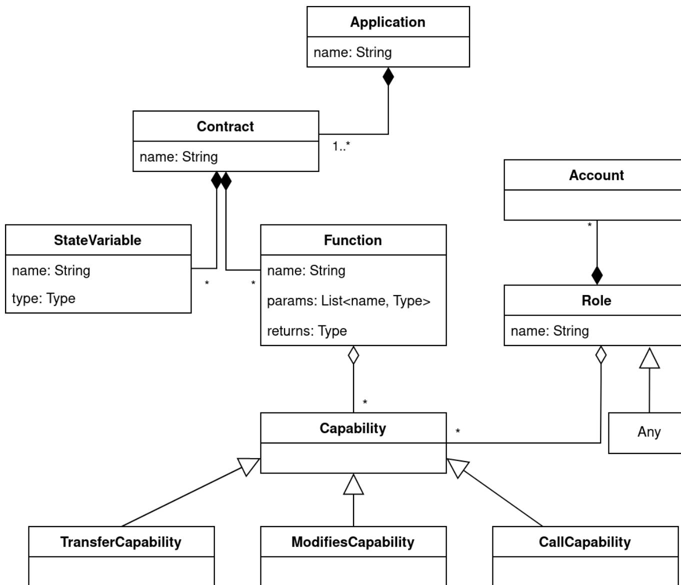

Figure 1. Smart Contract Application Metamodel

a) Applications and contracts: For our purposes, a smart contract application is a set of smart contracts which are implemented to serve a specific purpose. Each contract in an application is either newly developed for the application, or it already exists and its implementation is known to the developer. This makes it possible to include existing library contracts in an application.

Thus, in our metamodel, an application is the top-level model element. It has a name and consists of one or more individual contracts. These, in turn, are named and consist of state variables and functions.

We call the set of contracts of an application $\mathcal{CS}$.

b) State variables: State variables consist of a name and a type; for this, we define a type system (cf. Figure 2). In order to allow modelling realistic and useful applications, our type system includes not only primitive types but also composite types which are commonly found in smart contract programming languages: arrays and mappings, as well as user-defined types, i.e., structs.

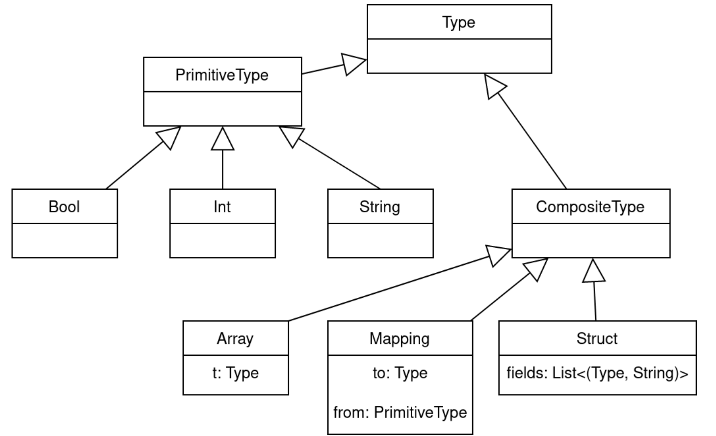

Figure 2. Type System

Array state variables consist of a name and the type of the array elements. Structs are defined by a list of tuples $(name, type)$ which define the struct's fields. Mappings have a key type and a value type. While the latter can be any type, the former must be a primitive type (cf. the implementation of mappings in Solidity, the most-used smart contract programming language).

The description of contracts and state variables gives rise to the set $\mathcal{L}$ of locations of an application. Intuitively, a location is either a variable or an element of a composite data type, i.e., an array element, a value in a mapping, or a struct field.

Formally, the set of locations of a smart contract application is

$$\begin{aligned}\mathcal{L} \quad ::= \ \mathcal{CS} \times \quad & (primitiveVarName \ \cup \\ & arrName \cup (arrName \times \mathbb{N}) \ \cup \\ & mapName \cup (mapName \times mapKey) \ \cup \\ & structName \cup (structName \times fieldName))\end{aligned}$$

where

- $\mathcal{CS}$ is the set of contracts of an application (cf. above),

- *primitiveVarName* is the set of names of all primitive state variables,
- *arrName* is the set of all array variable names,
- *mapName* is the set of all mapping variable names,
- *mapKey* is the set of all possible mapping keys,
- *structName* is the set of all struct variables, and
- *fieldName* is the set of all struct field names.

With this, we can also define the state of an application.

Definition 1 (State): The state of an application $\mathcal{S} : \mathcal{L} \rightarrow \mathcal{V}$ is a function that assigns values to locations, where the set $\mathcal{V}$ contains all primitive types as well as references (i.e., variable names).

c) Functions: Functions enable a user's interaction with a smart contract application. For the purpose of this work, we say that functions consist of a name, a set of parameters, and a set of (named) return types. Functions can also be annotated with capabilities (see Section IV-C).

We say that $\mathcal{F}$ is the set of all functions of an application. Each element f in $\mathcal{F}$ is of the form *(contractName, funIdentifier)*, i.e., it consists of the name of the contract and the name of the function.

B. Actors

Going forward, our model provides elements to describe and summarize the entities in a smart contract application which possess agency, i.e., which are able to call functions. We call such an entity an actor, and call the set of all actors $\mathcal{A}$.

In smart contract platforms, the entities that can call functions are usually called accounts. They are uniquely identified by an *accountID*, e.g., the address on the Ethereum platform. Whether an account represents a program or a person is not relevant for our purposes.

The set of all accounts is called $\mathcal{ACC}$.

In order to make modelling accounts feasible, we allow summarizing them in the form of roles. In our metamodel, a set of roles can be attached to an application. In this paper, we do not go into the details of role-based access control (some approaches have been proposed by Chatterjee et al. [3] and Töberg et al. [17]). We simply assume that the implementation of the application contains a boolean function *hasRole(Account a, Role r)* which returns true iff Account a indeed has role r.

In order to allow functions which are not access controlled at all, we define a role *any* which exists in every application.

The set of roles of an application is $\mathcal{R}$. Each $r \in \mathcal{R}$ defines a set $R_r \subseteq \mathcal{ACC} := \{a \in \mathcal{ACC} \mid hasRole(a, r)\}$.

Apart from accounts, functions themselves can also call other functions. Therefore, we include them in the set of actors, which is defined as $\mathcal{A} := \mathcal{ACC} \cup \mathcal{F}$.

C. Resources and Capabilities

Motivated by our perspective of smart contracts described in Section III, we identify three important resources on smart contract platforms: The state, which represents assets on the ledger; functions, which manage and grant access to the state; and cryptocurrency, which has a role similar to actual currency and can be transferred via built-in primitives.

Table I
CAPABILITIES SYNTAX

CallCapability	::=	`calls` *funIDList* | `any` | `external`
funIDList	::=	*funID [, funID]**
ModifyCapability	::=	`modifies` *Locset*
Locset	::=	*LocExpr [, LocExpr]**
LocExpr	::=	*contractName . LocalLocExpr*
LocalLocExpr	::=	*primitiveVarName* | *CompositeVarName* [*arraySuffix* | *mapSuffix* | *structSuffix*]?
arraySuffix	::=	`[`*intExpr*`]` | `[`*intExpr .. intExpr*`]` | `[*]`
mapSuffix	::=	`[`*mapKey*`]` | `[*]`
structSuffix	::=	. *structMember* | `.*`
TransferCapability	::=	`transfers` *transfer [, transfer]**
transfer	::=	`(` *recipientSet* `,` *limitExpr* `)`
recipientSet	::=	`self` | `any` | *accountID* | *roleID*

From this, we identified three types of capabilities: Changing state, calling functions, and transferring currency. In our model, capabilities are assigned to actors, i.e., to roles and to functions. In this section, we introduce a grammar for defining capabilities. It is shown in Table I.

a) Calling functions: Roles and functions can be annotated with a list of functions they are allowed to call, following the `calls` keyword. There are two special values: `external` for declaring that the function may make external calls to functions that are not known at design time, and `any` for stating that a function may call any other function, including external functions.

b) Modifying State: The capability to modify state is described after the `modifies` keyword, which is followed by a list of location sets. These include variables of primitive type, but also composite variables and their elements, e.g., individual struct fields and array or mapping elements. Furthermore, array ranges can be specified, and the developer can also express that all elements of a composite type variable can be changed (but not the reference to the variable itself).

c) Transferring Currency: Transfer capabilities are initialized with the `transfers` keyword. They consist of two parts: A set of allowed recipients, and an expression that limits the amount of currency being sent. The recipients can be a specific account, but also the special value *self*, describing the caller of the function. This enables a developer to specify a recurring pattern of smart contract applications: A caller may initiate a transfer to themselves (e.g., by withdrawing money they deposited earlier), but not to anyone else. Furthermore, the set of allowed recipients can be described by a role. Finally, the `any` keyword expresses that there is no limitation as to the recipient of the money.

The limit is specified as a general integer expression. It

can be an integer literal, but it can also contain arithmetic or make reference to the state of the application, or to the value of function parameters at call-time. This enables the developer to specify limits that depend on the current state, e.g., limiting a withdrawal to the deposited amount.

d) Example: We provide a simple running example to illustrate the presented concepts. Our example application is a bank, where customers can deposit cryptocurrency and withdraw it at a later point. Furthermore, the owner of the bank can close it, thereby sending all withdrawable funds to the customers.

The example application is presented in Listing 1 in a simple XML-like syntax. Model elements from the overall smart contract application metamodel (Section IV) are in italics. Capability keywords are written in bold font.

The application consists of only one contract, which has two state variables (the `balances` mapping storing the customer balances, and `totBal` storing the overall balance of the contract) and three functions (`close()`, `deposit`, and `withdraw`). Furthermore, the application has two roles: An owner, and customers.[1] Roles and customers are annotated with capabilities, according to the grammar defined in Table I. The owner may call the `close` function and transfer currency to other accounts if they have the customer role. They may also modify all elements of the `balances` mapping as well as the `totBal` value. Customers, in turn, can call the `withdraw` function and access the value of the `balances` mapping, but only at their own address. They can also only initiate transfers to themselves, capped at their current balance. The application's functions are also given capabilities: `close` may transfer money to all customers, limited only by the contract's total balance, and it may modify all state variables of the application. The `deposit` function is marked as callable by everyone. The `withdraw` function can transfer currency, but only to the caller of the function (denoted by the `self`) keyword).

```
application: bank
roles:
  role: any
    calls Bank.deposit()
  role: owner
    calls Bank.close
    transfers (customer, Bank.totBal)
    modifies Bank.balances[*], Bank.totBal
  role: customer
    calls Bank.withdraw
    transfers (self, Bank.balances[self])
    modifies Bank.balances[self], Bank.totBal
contracts:
  Bank:
    stateVariables:
      balances: mapping(account => int)
      totBal: int
    functions:
      close() returns ()
        transfers (customer, Bank.totBal)
        modifies Bank.balances[*], Bank.totBal
      deposit(int amt) returns ()
        callableByAny
        modifies Bank.balances[self], Bank.totBal
      withdraw(int amt) returns ()
        transfers (self, Bank.balances[self])
        modifies Bank.balances[self], Bank.totBal
```

Listing 1. Example: A simple bank application

V. Semantics of Capabilities

In this section, we give a formal meaning to the capability syntax given in Table I.

For each kind of capability, we define an evaluation function which maps syntactical elements to an abstract set of capabilities.

For this, we define an auxiliary function *get:* $String \Rightarrow \mathcal{CSF} \cup \mathcal{V} \cup \mathcal{ACC}$, which maps function identifiers, variable names and account identifiers to the corresponding elements in the sets $\mathcal{F}$, $\mathcal{V}$ and $\mathcal{ACC}$.

Furthermore, some capability descriptions may contain references to the state of an application, or to function parameters. Also, the identity of the caller may be significant to evaluate a capability description. All of these depend on the context of a function call. Therefore, we define a function *ctx* which evaluates a given expression in the context of a function call.

[1] A suitable implementation of the `hasRole()` function for this application is as follows: An account has the *owner* role if it is the one which created the contract. An account has role customer if `balances(a)` > 0.

Table II
Call Capability Evaluation

$[\![c_1.f_1, ..., c_n.f_n]\!]_C$	$::=$	$\{(get(c_1), get(f_1)), ..., (get(c_n), get(f_n))\}$
$[\![\texttt{external}]\!]_C$	$::=$	*external*
$[\![\texttt{any}]\!]_C$	$::=$	$\mathcal{F} \cup$ *external*

A. Call Capability

Definition 2 (Call Capability Evaluation): The call capability evaluation

$$[\![\cdot]\!]_C : FunctionIdentifiers \Rightarrow \mathcal{F}$$

is defined by the rules shown in Table II.

A given list of functions simply evaluates to the set of corresponding functions in $\mathcal{F}$. The `external` keyword evaluates to the corresponding set *external* of all functions which are not described in the application. While nothing is known about the behaviour of these functions, it is sensible to make the developer explicitly model external calls. Furthermore, if a function should be unrestricted, the developer can use the `any` keyword to signal this.

B. Modification Capability

Definition 3 (Modification Capability Evaluation): The modification capability evaluation $[\![\cdot]\!]_M : LocSet \Rightarrow \mathcal{L}$ is defined by the rules shown in Table III.

A list of location expressions is evaluated to the disjunction of the location sets described by each expression. Variable names evaluate to the locations of these variables. For arrays, specifications of single indices or index ranges can contain integer expressions. These are evaluated under the context of the function call. For mappings and structs, the $*$ operator

evaluates to all possible mapping keys or struct fields, respectively.

C. Transfer Capability

Definition 4 (Transfer Capability Evaluation): The transfer capability evaluation $[\![\cdot]\!]_T^{ctx}$: *transfer* $\Rightarrow (\mathcal{ACC} \times \mathbb{N})^2$ is defined by the rules shown in Table IV.

Transfer capabilities consist of two parts, with the first specifying the set of allowed recipients, and the second setting an upper bound for the amount of transferred currency. For the recipients, the `self` keyword evaluates to the caller of the function (retrieved by the *get* helper function). A specific account identifier evaluates to that account, and `any` evaluates to the set of all accounts. A role identifier evaluates to the set of all accounts who, in the current context, have the indicated role.

VI. Analysing Model Consistency

In this section, we define what constitutes a consistent model. Intuitively, a consistent model is one where capabilities cannot be violated. A violation on the model level occurs, e.g., if a function has a less restrictive capability than an account which is allowed to call that function. In this section, we develop an analysis to decide whether a model is consistent. Additionally, we want to ensure that the specified privileges are tight in comparison to the required capabilities.

Due to our semantics definition, the model consistency is a simple subset relationship of the capabilities. Let $\mathcal{R}$ be the set of roles, $\mathcal{F}$ the set of functions and $\mathcal{A} = \mathcal{R} \cup \mathcal{F}$ the set of all actors defined for an application (as above).

Definition 5 (Model consistency): A model is consistent iff

$$\forall a \in \mathcal{A} : \forall f \in \mathcal{C}_{call}^a \forall ctx : [\![\mathcal{C}(f)]\!]_\gamma^{ctx} \subseteq [\![\mathcal{C}(a)]\!]_\gamma^{ctx} \ ,$$

where $\mathcal{C}_{call}^a$ denotes the set of functions specified as callable by a role a, $\gamma \in \{C, T, M\}$ is the domain of call, transfer, or modification capabilities, and ctx an arbitrary context for the evaluation of expressions.

Note that the consistency notion is also transitive, in the sense that if a function f_1 transitively calls f_3 via f_2, the specified capabilities of f_1 need to be a superset of the specified capabilities of f_3:

$$[\![\mathcal{C}(f_3)]\!]_\gamma \subseteq [\![\mathcal{C}(f_2)]\!]_\gamma \subseteq [\![\mathcal{C}(f_1)]\!]_\gamma$$

The call capabilities are directly checkable, since they are a set of symbols. In contrast, the modification and transfer capabilities contain symbolic integer expressions over the set of states and parameters. Therefore, further reasoning is needed to decide whether a set of modification capabilities is more restrictive than another. The same is true for transfer capabilities.

For both kinds of capabilities, we derive proof obligations in the following definitions which are automatically checkable with SMT solvers, e.g. Z3 [**?**], due to their support of integer theory.

Definition 6: Given two sets M_1, M_2 of modify capabilities, we say M_1 is a subset of M_2 ($M_1 \subseteq_{\text{modify}} M_2$) iff the following formula is valid

$$\bigwedge_{m_1 \in M_2} \bigvee_{m_2 \in M_2} m_1 \leq m_2 \ ,$$

where $m_1 \leq m_2$ is defined as

$$m_1 \leq m_2 := \begin{cases} \mathit{true} : \text{if } m_1 = m_2 \\ \forall \underbrace{x_1, \ldots, x_n}_{x_i \in FV(e_1)} : \exists \underbrace{y_1, \ldots, y_n}_{y_i \in FV(e_2)} : \\ \qquad e_1 \geq 0 \land e_2 \geq 0 \rightarrow e_1 = e_2 \\ \qquad\quad : \text{if } m_1 = a[e_1] \text{ and } m_2 = a[e_2] \\ \mathit{false} : \text{otherwise} \end{cases}$$

The expressions e_1 and e_2 are the integer expressions describing the set of allowed indices.

$FV(e)$ denotes the free variable of an expression e. These arise from the context of a call, i.e., the state and the parameters of the function. Since this context is not known on the model level, we abstract from the free variables in an over-approximated manner. The definition $m_1 \leq m_2$ evaluates to true for symbolically identical capabilities, e.g., if both capabilities allow the modification of the same state variables.

For accesses of array indices, e.g., $m[4 * x] \leq m[2 * x]$, we need to compare the described set of indices for all possible positive indices: every index of the more or equally restrictive expression must be possible in the other expression. In the example the expression $4 * x$ is more restrictive than $2 * x$ as it describes fewer indices, but not vice versa.

For transfer capabilities, the consistency definition is as follows:

Definition 7: Given two transfer capabilities $t_1 = (rec_1, am_1)$ and $t_2 = (rec_2, am_2)$, we say t_1 is more restrictive than t_2 (denoted $t_1 \leq t_2$) iff

- receiver set rec_1 is a subset of rec_2, and
- the value of am_1 is lower than am_2 under all interpretations of the free variables in am_1 and am_2:

$$\forall \underbrace{x_1, \ldots, x_n}_{x_i \in FV(am_1) \cup FV(am_2)} : rec_1 \subseteq rec_2 \land am_1 \leq am_2$$

For a two sets T_1, T_2 of transfer capabilities, we say T_1 is a subset of T_2 iff

$$\forall t_1 \in T_1 : \exists t_2 \in T_2 : t_1 \leq t_2 \ .$$

The current consistency definitions only specify that the permissions of an actor must not be stricter than those of the functions they are allowed to invoke. Moreover, we might want to ensure that an actor only has the necessary the capabilities to work properly. This *principle of least privilege* is similar to consistency but investigates the opposite direction. For an actor $a \in \mathcal{A}$, we say it fulfills its principle of least privilege iff

$$\forall c \in [\![\mathcal{C}(a)]\!]_\gamma : \exists f \in \mathcal{C}_{call}^a : c \in [\![\mathcal{C}(f)]\!]_\gamma$$

is valid. The definition requires that every capability of the actor a is required for at least one specified callable function.

Table III
MODIFY CAPABILITY EVALUATION

$[\![l_1, l_2, ...]\!]_M$	::=	$[\![l_1]\!]_M \cup [\![l_2]\!]_M \cup ...$
$[\![cName, locExpr]\!]_M$	::=	$\{get(cName)\} \times [\![locExpr]\!]_M$
$[\![primitiveVarName]\!]_M$	::=	$\{get(primitiveVarName)\}$
$[\![ArrayVarName]\!]_M$	::=	$\{get(ArrayVarName)\}$
$[\![ArrayVarName\ arraySuffix]\!]_M$	::=	$\{get(ArrayVarName)\} \times [\![arraySuffix]\!]_M$
$[\![MappingVarName]\!]_M$	::=	$\{get(MappingVarName)\}$
$[\![MappingVarNamemappingSuffix]\!]_M$	::=	$\{get(MappingVarName)\} \times [\![mappingSuffix]\!]_M$
$[\![StructVarName]\!]_M$	::=	$\{get(StructVarName)\}$
$[\![StructVarNamestructSuffix]\!]_M$	::=	$\{get(StructVarName)\} \times [\![structSuffix]\!]_M$
$[\![\texttt{[intExpr]}]\!]_M$	::=	$\{ctx(intExpr)\}$
$[\![\texttt{[intExpr1 .. intExpr2]}]\!]_M$	::=	$\{i \in Int \mid ctx(intExpr1) \leq i \wedge i \leq ctx(intExpr2)\}$
$[\![\texttt{[*]}]\!]_M$	::=	$Int \cup MapKeySet$
$[\![\texttt{[.*]}]\!]_M$	::=	$StructFieldSet$
$[\![mapKey]\!]_M$	:=	$\{mapKey\}$
$[\![.structMember]\!]_M$	::=	$\{structMember\}$

Table IV
TRANSFER CAPABILITY EVALUATION

$[\![t_1, t_2...]\!]_T^{ctx}$	::=	$[\![t_1]\!]_T^{ctx} \cup [\![t_2]\!]_T^{ctx} \cup ...$
$[\![rec, limit]\!]_T^{ctx}$	::=	$[\![rec]\!]_{rec}^{ctx} \times \{n \in \mathbb{N} \mid n \leq [\![limit]\!]^{ctx}\}$
$[\![\texttt{self}]\!]_{rec}^{ctx}$	::=	$\{ctx(self)\}$
$[\![\texttt{any}]\!]_{rec}^{ctx}$	::=	$\mathcal{ACC}$
$[\![accountID]\!]_{rec}^{ctx}$	::=	$\{get(accountID)\}$
$[\![roleID]\!]_{rec}^{ctx}$	::=	$\{a \in \mathcal{ACC} \mid ctx(hasRole(a, get(roleID)))\}$

As this notion also relies on the model-level specification of the callable functions, it might also be an over-approximation in contrast to the actual called function of the implementation.

VII. A SECURE SOLIDITY IMPLEMENTATION

While Section VI describes how to achieve a consistent model of a smart contract application with a formally specified security policy, this section gives a definition of what properties an implementation needs to fulfill to be considered conformant. Based on this definition, we sketch how to arrive at an implementation in the Solidity programming language that is conformant w.r.t. the model according to the definition.

For this, we define that an implementation is conformant to its model iff

Definition 8 (Implementation conformance): We say that an implementation is conformant to its model iff

1) for each function $f \in \mathcal{F}$, an account $a \in \mathcal{ACC}$ only has access to f if a has a role $r \in \mathcal{R}$ s.t. $f \in \mathcal{C}_{call}^r$
2) each function f conforms to its capability specification, i.e.
   a) f calls only functions g where $g \in \mathcal{C}_{call}^f$
   b) during any execution of f, any location l where $l \notin \mathcal{C}_{state}^f$ remains unchanged
   c) f only makes transfers $(to, amt) \in \mathcal{C}_{transfer}^f$

Whether an implementation conforms to its model depends on the platform where it is implemented. Therefore, we sketch a process for the Solidity programming language of the Ethereum platform, although the general ideas might apply for any platform.

Solidity has several characteristics and mechanisms relevant to developing a secure application in our proposed process. First, in Ethereum, accounts are identified by addresses in the form of 160-bit integers. Second, Solidity provides the `requires` keyword, which checks a boolean condition at runtime and reverts if the condition is not met. Furthermore, Solidity has *modifiers*, which wrap functionality (e.g., parameter checks) and can be added as a keyword to a function header. Both the `requires` mechanism and modifiers can be applied to implement access control.

As a basis for generating the proof obligations described in Section VI and the source code stubs as described below, we developed a Scala implementation of our metamodel and the capability grammar. After arriving at a consistent model (cf. Section VI), a smart contract application developer can achieve a conformant implementation via a combination of code and annotation generation on the one hand, and formal methods and static analysis on the other.

A. Code Generation

From the model, our code generator generates source code stubs as follows: First, we generate a `contract` file for each contract in $\mathcal{CS}$. The file contains a variable declaration for each state variable, and a function header consisting of name, parameters and return type for every function. Then, we generate a smart contract which is responsible for access control to functions. It contains an enum of the roles specified in the model, and a function `hasRole(Role r, address a)`, which can be called to check at runtime whether a given address has a certain role.

Furthermore, we generate access control modifiers: For each function $f \in \mathcal{F}$, we compute the set $r_f \subseteq \mathcal{R}$ of roles that may access f. Then we generate a modifier which checks whether a given account with address a has any of the roles in r_f.

```
modifier onlyOwner {
  require(hasRole(msg.sender, Roles.OWNER));
  _;
}

function close() onlyOwner {
  ...
}
```

Listing 2. Example access control modifier

```
function wrappedTfWithdraw(address a, uint amt)
    internal {
  require(a == msg.sender);
  require(amt <= Bank.balances[msg.sender])
  a.transfer(amt);
}
```

Listing 3. Example Wrapped Transfer Function

Of course, if the access sets of two functions are equal, the corresponding modifier only needs to be generated once. An example is shown in Listing 2, where a modifier is defined to check that the caller has the Owner role, and the `close` function is generated with this modifier in the header.

Furthermore, for each transfer capability $t \in \mathcal{C}^f_{transfer}$ of a function f, we generate a *wrapped transfer function* which ensures at runtime that the function adheres to its capabilities. The function is internal (i.e., it can only be called from within the contract) and takes an address *addr* and an amount *amt* as a parameter. For a capability *t* consisting of a recipient set expression *rs-exp* and a limit expression *limit-expr*, the wrapped function is implemented as follows: At first, it checks whether the address parameter matches the recipient set. This is done by a `require` statement, with a condition depending on *rs-expr*: If *rs-expr* is an address, it must be equal to *addr*. If it is `self`, *addr* must be equal to `msg.sender`.[2] If *rs-expr* is a role *r*, it is checked whether `hasRole(addr, r)` returns true. Finally, if *rs-expr* is `any`, the check is omitted. Afterwards, it is checked whether *amt* is less than or equal to *limit-expr*.

We give an example in Listing 3. The function `withdraw()` of our running bank example has a capability of sending currency, but only to the caller, and the amount of currency is limited by the balance of the caller. By using the generated function `wrappedTfWithdraw`, the developer can be sure that the implementation adheres to the model.

B. Formal Analysis

While transfers and access of accounts to functions can be handled by code generation in a correct-by-construction manner, this is not possible for ensuring that a function only calls the functions it is allowed to call, and only modifies those parts of the application's state specified in the model. Checking these properties is only possible on the finished implementation. It can be done using static analysis and formal verification tools.

For analysing whether all functions conform to their call capabilities, we developed a simple static source code analysis based on a publically available Solidity grammar.[3] For every function *f*, our analysis collects all function calls that occur explicitly, as well as all occurrences of the `call` and `callcode` keywords and their parameters. It then compares the names of the called functions to the functions in $\mathcal{C}^f_{call}$. If $\mathcal{C}^f_{call}$ contains the *external* value, then only functions within the application itself are forbidden. If the capability specification contains the `any` keyword, the analysis is skipped.

If the analysis finds a function call that is not allowed per capability specification, it fails. This is a deliberate over-approximation, as our analysis will flag some legitimate calls as not allowed (for example, in Solidity, functions can be passed as parameters and assigned to variables and then called). However, it is easy to write the implementation in a way that satisfies the analysis, and we think application developers will benefit from the increased clarity.

For analysing whether all functions adhere to their modification capabilities, we have to prove that a function modifies only those locations specified in its capabilities. Formally, we need to prove that

$$\forall f \in \mathcal{F}: \ \forall l \in \mathcal{C}^f_{state} : \mathcal{S}^{pre}_f(l) = \mathcal{S}^{post}_f(l)$$

where $\mathcal{S}^{pre}_f$ and $\mathcal{S}^{post}_f$ are the states before and after the execution of f.

For this, we employ automated generation of formal specification in combination with the SOLC-VERIFY formal verification tool [7]. SOLC-VERIFY takes as an input solidity source code that is annotated with formal specification, such as invariants and function contracts. Function contracts consist of a pre- and postcondition, but can also include a *frame condition*, i.e., a statement about what parts of the state a funtion is allowed to modify.

We utilize SOLC-VERIFY as follows: During code generation (cf. Section VII-A), we annotate every function with one frame condition per *LocExpr* in its modification capability specification. The annotations are in SOLC-VERIFY's annotation language. An example is shown in Listing 4: The `withdraw()` function is annotated with frame conditions which state that the function may only modify the caller's own `balances` mapping element and the overall balance of the contract. This corresponds to the modification capabilities defined for the function in the example (Listing 1). If the SOLC-VERIFY tool successfully proves that a given implementation adheres to this specification, then it follows that the function adheres to its capability specification.

After the implementation is finished, the developer conducts a formal proof of correctness with SOLC-VERIFY. If the proof succeeds, this means that all functions adhere to their modification capabilities.

[2]Note that because the wrapped function is marked as `internal`, the `msg.sender` variable is passed on to it from the calling function.

[3]https://github.com/solidityj/solidity-antlr4/blob/master/Solidity.g4

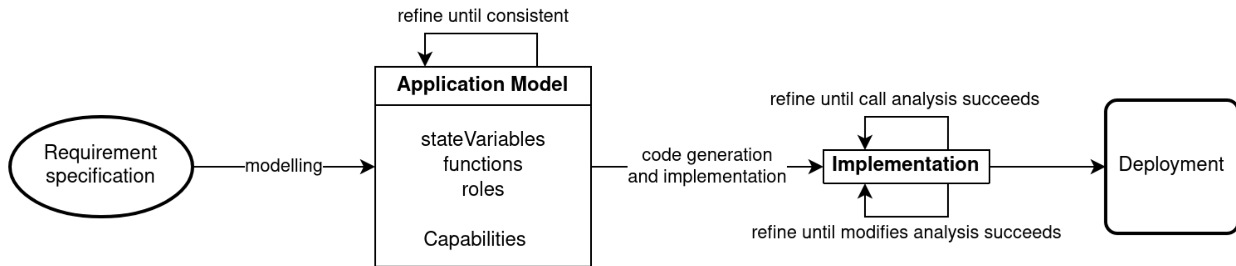

Figure 3. The overall development process of a secure smart contract application

```
/// @notice modifies balances[msg.sender]
/// @notice modifies totBal
function withdraw(uint amt) onlyCustomer {
  ...
}
```

Listing 4. Example SOLC-VERIFY Frame Condition

Note that, again, this is an over-approximation: It is possible that the proof of correctness of the frame conditions does not succeed although the implementation is actually correct w.r.t. the modification capabilities. This can happen, e.g., if auxiliary specification such as loop invariants are missing or not sufficient.

As an alternative to our over-approximating approach, a developer could also conduct a different static analysis, such as an information flow analysis. SLITHER [5] is one of several tools which have been developed for this purpose. It automatically analyses a given application statically and can be configured to output the call graph and all variables written by a certain entry point (e.g., a function). An alternative to our approach would be to run SLITHER in this way and inspect its output. If the callgraph for a function contains functions that are not contained in $\mathcal{C}^f_{call}$, or if a function writes a variable which is not in $\mathcal{C}^f_{state}$, then the function does not adhere the capabilities specified in the model, and its implementation needs to be corrected.

One drawback of using SLITHER or comparable static analysis tools is the possibility of false positives and false negatives. With our analysis and SOLC-VERIFY's proof of frame conditions, a developer can be sure that an implementation is correct; SLITHER gives no such guarantee. On the other hand, a formal proof of correctness can require developer involvement. For example, it can be necessary to provide auxiliary specification, e.g., loop invariants, for an automated proof to succeed. The decision for a specific analysis tool depends on the specifics of the application (or even individual functions) and has to be made on a case-by-case basis.

C. Correctness

At the beginning of this section, we gave a definition of when we consider an implementation to be conformant to a model. We argue that the process sketched in this section leads to an implementation which is correct in that sense, if the following assumptions hold: The developer does not change the generated code, but only adds the function bodies; they only use the auto-generated wrapped transfer methods instead of Solidity's built-in transfer methods; and all source code analyses (cf. above) correctly return a successful result (e.g., a proof that the implementation adheres to the generated frame annotations).

If these assumptions hold, condition 1) is guaranteed by the access control modifiers. Condition 2a) and 2b) are guaranteed by our static call analysis and by SOLC-VERIFY's proof that all the generated frame conditions hold. Finally, condition 2c) holds because the generated transfer funtions ensure at runtime that all functions adhere to their specified transfer capabilities.

VIII. Development Process Overview

In this final section, we give an overview of the intended process that a developer is supposed to follow in our approach. Figure 3 illustrates this process.

At the beginning, there is a requirement specification which contains a security policy. From this specfcation, a model - an instantiation of the metamodel developed in this work - is derived, with capabilities according to the security policy. The developer then performs analyses as described in Section VI on the model, refining it until it is consistent (and, ideally, the capabilities are precise, i.e., no actor has greater capabilities than it needs).

From this model, the developer uses the code generator provided by us to generate the code skeleton and the access control smart contract (Section VII-A). They implement the access control contract according to the role model specified in the security policy. Furthermore, they implement all functions (using the generated wrapper transfers for currency transactions). Then, the implementation is analysed with our static call analysis, and SOLC-VERIFY is employed to find a proof that the generated frame conditions hold (Section VII-B). The implementation is refined until these analyses succeed. When that is the case, the application can be deployed.

IX. Conclusion and Future Work

In this work, we propose a solution for developing secure smart contract applications from existing requirements specifications. We argue that implementing a security policy directly in source code is error-prone and likely leading to vulnerabilities. Therefore, we develop a metamodel of smart contract applications with an attached capability-based

security specification. This makes it very easy for smart contract developers to abstract away from the complexity of the source code and focus on the security-relevant aspects of an application.

Going forward, we define a notion of consistency and precision on our model, based on set-theoretic semantics. We also develop analyses to check whether a given model conforms to these notions. Furthermore, we describe how to turn a model into a conformant implementation. This is done by a combination of code generation and simple static analysis that we implemented on the one hand, and existing formal verification tools on the other hand.

A natural way of extending our methodology is integrating analysis tool results into the model automatically. We are currently working on a method to parse custom output of the Slither tool. This will not only enable direct and helpful feedback in case of non-conformant implementation, but might also help automating the entire development process while retaining the security guarantees.

Furthermore, we plan on extending our capability model by adding conditional capabilities, i.e., capabilities that are only in effect if some condition, such as a time constraint, is met.

Our metamodel is platform-independent, and our methodology can easily be adapted for other platforms or other programming languages. In particular, our approach may be used for programming languages which have built-in concepts of access control and/or resource management, or tooling which supports these concepts. As an example, it would be possible to derive an implementation in the Vyper language, and ensure conformance to transfer capabilities with support of the 2vyper verification tool.

So-called permission-based blockchain platforms like Hyperledger Fabric [1] do not fully match our notion of smart contracts, since they enforce a closed world in which all participants are known and identifiable. This allows defining role models and security policies which can be implemented above the source code level, partly negating the advantage that our modelling approach brings. However, the methods we developed for analysing a given model can still be used to detect inconsistency or imprecision in the capability definitions of a Fabric application. This raises the confidence in the security of the application, and may shift the detection of errors from runtime to design time, thereby lowering the cost of fixing them.

References

[1] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S.W., Yellick, J.: Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In: Proceedings of the Thirteenth EuroSys Conference. pp. 30:1–30:15. EuroSys '18, ACM, http://doi.acm.org/10.1145/3190508.3190538

[2] Bräm, C., Eilers, M., Müller, P., Sierra, R., Summers, A.J.: Modular verification of collaborating smart contracts abs/2104.10274

[3] Chatterjee, A., Pitroda, Y., Parmar, M.: Dynamic Role-Based Access Control for Decentralized Applications. In: Chen, Z., Cui, L., Palanisamy, B., Zhang, L.J. (eds.) Blockchain – ICBC 2020. pp. 185–197. Lecture Notes in Computer Science, Springer International Publishing

[4] Dharanikota, S., Mukherjee, S., Bhardwaj, C., Rastogi, A., Lal, A.: Celestial: A Smart Contracts Verification Framework. In: 2021 Formal Methods in Computer Aided Design (FMCAD). pp. 133–142

[5] Feist, J., Grieco, G., Groce, A.: Slither: A Static Analysis Framework for Smart Contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 8–15

[6] Frantz, C.K., Nowostawski, M.: From Institutions to Code: Towards Automated Generation of Smart Contracts. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 210–215

[7] Hajdu, A., Jovanovic, 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

[8] Jurgelaitis, M., Čeponienė, L., Butkus, K., Butkienė, R., Drungilas, V.: MDA-Based Approach for Blockchain Smart Contract Development 13(1), 487, https://www.mdpi.com/2076-3417/13/1/487

[9] Kolb, J., Yang, J., Katz, R.H., Culler, D.E.: Quartz: A framework for engineering secure smart contracts

[10] Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc.

[11] Levy, H.M.: Capability-Based Computer Systems. Butterworth-Heinemann

[12] Mueller, B.: Smashing Ethereum Smart Contracts for Fun and Real Profit, https://github.com/muellerberndt/smashing-smart-contracts/blob/0663ad015b0a6ce08053d48731cdee1e7bc4e726/smashing-smart-contracts-1of1.pdf

[13] Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: Safety Verification of Smart Contracts. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1661–1677

[14] Skotnica, M., Pergl, R.: Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts. In: Aveiro, D., Guizzardi, G., Borbinha, J. (eds.) Advances in Enterprise Engineering XIII. pp. 149–166. Lecture Notes in Business Information Processing, Springer International Publishing

[15] Stephens, J., Ferles, K., Mariano, B., Lahiri, S., Dillig, I.: SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 555–571

[16] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical Security Analysis of Smart Contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82. CCS '18, Association for Computing Machinery, https://doi.org/10.1145/3243734.3243780

[17] Töberg, J.P., Schiffl, J., Reiche, F., Beckert, B., Heinrich, R., Reussner, R.: Modeling and Enforcing Access Control Policies for Smart Contracts. In: 2022 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS). pp. 38–47

[18] Xu, W., Fink, G.A.: Building Executable Secure Design Models for Smart Contracts with Formal Methods. In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 154–169. Lecture Notes in Computer Science, Springer International Publishing

[19] Zhu, J., Hu, K., Filali, M., Bodeveix, J.P., Talpin, J.P., Cao, H.: Formal Simulation and Verification of Solidity contracts in Event-B. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). pp. 1309–1314