

SMACS: Smart Contract Access Control Service

Bowen Liu*, Siwei Sun[†], Pawel Szalachowski*

*Singapore University of Technology and Design, Singapore

[†]State Key Laboratory of Information Security, Institute of Information Engineering,
Chinese Academy of Sciences, Beijing 100093, China

Abstract—Although blockchain-based smart contracts promise a “trustless” way of enforcing agreements even with monetary consequences, they suffer from multiple security issues. Many of these issues could be mitigated via an effective access control system, however, its realization is challenging due to the properties of current blockchain platforms (like lack of privacy, costly on-chain resources, or latency). To address this problem, we propose the SMACS framework, where updatable and sophisticated *Access Control Rules (ACRs)* for smart contracts can be realized with low cost. SMACS shifts the burden of expensive ACRs validation and management operations to an off-chain infrastructure, while implementing on-chain only lightweight token-based access control. SMACS is flexible and in addition to simple access control lists can easily implement rules enhancing the *runtime* security of smart contracts. With dedicated ACRs backed by vulnerability-detection tools, SMACS can protect vulnerable contracts *after deployment*. We fully implement SMACS and evaluate it.

Index Terms—Blockchain; Smart Contract; Access control; Ethereum; Runtime verification

I. INTRODUCTION

Blockchain-based platforms like Ethereum [1] have made the concept of self-enforcing smart contract [2] into reality. A smart contract is a special computer program that executes on the global virtual machine running upon the distributed and decentralized ledger. By running a consensus protocol and following the replicated state machine model a unified view of the system state over all network participants is imposed.

Like all computer programs, it is likely that most non-trivial smart contracts will contain errors [3], [4], [5]. These smart contract related errors should be addressed even more seriously than ordinary program bugs. Firstly, smart contracts are often deployed over transparent and permissionless blockchain platforms, thus anyone can inspect and interact with them. Secondly, due to its immutability, it is hard to upgrade or simply “kill” an already-deployed smart contract when attacks are discovered as the contract could have become an important part of the ecosystem (i.e., other contracts hardcode its address). Finally, smart contracts determine how units of value convertible to real money move, making them a high-value target with intrinsic economic incentives. In the past few years, several hundreds of millions worth of USD were stolen or frozen due to flawed smart contracts [6], [7]. For instance, the infamous attack on the TheDAO [5] smart contract resulted in over 50 million US Dollars worth of Ether were drained at the time the attack occurred. Given the severity of the attack, the Ethereum community finally agreed on hard-forking.

As a consequence, the community has made a great effort on developing methodologies and tools to ensure the security of

smart contracts. One line of research focuses on the security analysis of smart contracts by verifying their code [8], [9], [10], [11]. However, most of these approaches are unable to protect deployed smart contracts. Another approach is to integrate runtime defensive mechanisms into the deployed smart contracts and their runtime environment. With this approach, security-risky interactions with a vulnerable smart contract can be detected and mitigated “on-the-fly” in runtime [12], [13]. These mechanisms usually require integration with the execution environment (i.e., with the virtual machine deployed) to be useful in production, but unfortunately that hinders their adoption. Ideally, a defensive mechanism with arbitrary complexity would be put into a smart contract itself, but in practice it is infeasible since on-chain resources are expensive. Moreover, such a mechanism would be difficult to manage and update, ACRs would be publicly visible, and available smart contract languages with virtual machines associated would limit its capabilities. For example, it would be very costly and inconvenient to enhance the security of smart contracts by encoding fine-grained ACRs into them, which is a fairly mature and traditional approach for centralized systems.

In this work, we propose the SMACS framework, a cost-effective access control service that is not simply a token-based authentication system but aims to enhance the *runtime security* of smart contracts. In SMACS, a smart contract only needs to perform lightweight token verifications, which introduces a clear on/off-chain sides separation with minimized on-chain trusted computing base. The on-chain storage and computation requirement are minimized by various techniques and by shifting the burden of access control and ACR management to an off-chain service. SMACS framework not only supports fine-grained and updatable ACRs, but also is easily extensible by integrating recently developed smart contract vulnerability detection tools and security enhancement mechanisms [14], [12]. The combination of SMACS with runtime verification tools is powerful as it provides the benefits of these tools immediately without requiring updating virtual machines by all blockchain participants. SMACS is deployable as today, does not require any blockchain platform changes, and by moving security rules off-chain preserves their privacy.

II. BACKGROUND AND MOTIVATION

A. Blockchain and Smart Contracts

In the last ten years, the blockchain technology initiated by the Bitcoin [15] cryptocurrency has fueled great

innovations. Relying on the consensus protocol behind Bitcoin, it is not long before Ethereum [1] was built – a general-purpose blockchain system well-beyond cryptocurrencies which can execute programs on blockchain. Similarly to Bitcoin, Ethereum introduces its native cryptocurrency – *ether*, which is also used to incentivize system participants.

Ethereum can be regarded as a decentralized and replicated state machine whose state is maintained as a proof-of-work blockchain. The state transition of Ethereum is processed by the so-called *Ethereum Virtual Machine* (EVM) executing programs called smart contracts written in a Turing-complete language. Due to the Turing completeness, one can implement self-enforcing smart contracts with nearly arbitrary logic. As a result, we have witnessed a wide range of applications of smart contracts in different domains [14], [16].

Smart contracts inherit some essential properties from its underlying blockchain. Once a smart contract is deployed on the Ethereum platform, its code is immutable and visible by every node in the Ethereum network, and all transactions calling it are also transparent to all. Therefore, it is the duty of the contract developer to implement proper access control or defensive mechanisms prior to deployment, and failed to do so can lead to tremendous and irreversible financial losses.

B. EVM and Solidity

EVM executes smart contracts as a Turing complete stack-based language at low level called Ethereum bytecode. In practice, smart contracts are typically developed in high-level languages such as Solidity, Vyper, Serpent, etc., and are compiled into bytecode by the corresponding compilers.¹ In this work we focus on Solidity as it is the most popular and developed language of Ethereum. There are three memory regions of a smart contract program: stack, memory, and storage. The *stack* and *memory* are volatile and cheap to use. The *storage* is maintained on blockchain and is the only persistent memory region across transactions.

The execution logic of a smart contract is modularized into methods which are the executable code segments within a contract [21]. Method calls can happen internally or externally and have different levels of visibility towards other contracts. There are four types of visibilities for methods in Solidity:

- **external** methods are part of the contract interface, which means they can be called from other contracts and via transactions. An external methods cannot be called internally,
- **public** methods are part of the contract interface and can be either called internally or via messages (see the next section),
- **internal** methods can only be accessed internally (i.e., from within the current contract or contracts deriving from it),
- **private** methods are only visible for the contract they are defined in and not in derived contracts.

All computational or memory utilization in Ethereum is charged in *gas*, which can be regarded as a separate virtual

currency with its own exchange rate against ether [22]. The gas system is essential to incentivize system participants and prevent denial-of-service attacks or inadvertently resource-devouring transactions. Performing computation or storing data objects of large size (e.g., access rules) can be gas-expensive. In fact, according to our experiment, creating even a simple whitelist with 10k addresses would cost around \$300. Also managing such a list would have a linear cost in the number of update operations.

C. Transactions and Message calls

In Ethereum, every state change of the global singleton state machine is ultimately due to a *transaction*, a signed data package originated from an externally owned account. Transactions are recorded on the blockchain and can move value from one account to another or/and trigger smart contract execution. User accounts and smart contract instances are uniformly identified by unique addresses.

Contracts can call other contracts or send ether to non-contract accounts by *message calls*, the virtual objects that are never serialized and exist only in the Ethereum execution environment. Every transaction consists of a top-level message call which in turn can create further message calls. It implies that from a simple transaction initiated by an externally owned account, a *call chain* of contract executions can be triggered. Solidity allows smart contracts to access some global objects and properties of the blockchain [23]. In the context of this work the following objects are relevant:

- **tx.origin** - the sender of the transaction for full call chain (a list of all called methods that a given transaction triggers),
- **msg.sender** - the sender of the message for the current call. Let us consider a call chain triggered by a transaction T originated from the externally owned account u , where T calls the contract A , and A calls another contract B . Then from B 's perspective the value of `msg.sender` is the address of A while `tx.origin` is the address of u ,
- **msg.sig** - the method identifier (encoded as the first four bytes of *calldata*). Ethereum tags identifiers of each method for every smart contract. When A calls $B.funcB(argA, argB)$, the value of `msg.sig` seen by B would be the identifier of `funcB()`,
- **msg.data** - the complete *calldata* (the method identifier and passed arguments). The value of `msg.data` for the case above is `msg.sig` appended with the encoded values of `argA` and `argB`.

Transactions are signed by their originators and before processing them in EVM their authenticity is validated by the Ethereum network. To prevent replay attacks each transaction has a nonce which also is validated by participants. However, nonces cannot be accessed by Solidity contracts.

D. Access Control in Smart Contracts

Access control is a security technique that regulates who has access to certain system resources. The intention of access control in Ethereum smart contracts is to restrict the access to contract functionalities according to suitable criteria. In § II-B,

¹Note that even a smart contract is semantically bug-free with respect to the underlying high-level language, compilers may introduce language-specific vulnerabilities into the system [6], [17], [18], [19], [20], which once again highlight the importance of runtime security analysis.

the programming language of smart contracts already has some features to facilitate a minimum level of access control. However, these features are limited, only defining access control rules for built-in methods.

In general, implementing access control for smart contracts based on a permissionless blockchain is difficult. A naive approach of putting all access control logic into a smart contract would violate privacy and consume a lot of expensive on-chain resources. On the other hand, due to the immutability nature of the underlying blockchain, it is difficult to update the ACRs if they are managed over blockchain. To the best of our knowledge, there is no generic framework in the literature realizing efficient and flexible access control for smart contracts. However, such a framework could be highly beneficial for the security of smart contracts and the ecosystem. Currently, many token sales allow only approved users to participate in a token sale or trade. To achieve *on-chain* access control, the owner of the token sale contract maintains a whitelist listing addresses of authorized users. When a user tries to access the contract, there is an access control check verifying whether the user is whitelisted. For example, the Bluzelle decentralized database has paid 9.345 ETH (11,949 USD at the time) just to whitelist 7473 users for their token sale [24]. Similarly, OpenZeppelin [25] provides templates like role-based access control or “proxy contracts”. Unfortunately, these solutions are intended for on-chain contract management, do not allow for flexible changes at runtime, and still have all other limitations of on-chain access control.

To further illustrate the motivations behind our system, we give several other examples with brief comments.

Example 1. *A service provider may want to create a smart contract whose methods can be called only by a dynamic set of addresses (e.g., employees or business partners).*

Example 2. *The owner of a contract may want her smart contract to block the access from a predefined set of addresses.*

Example 1 and Example 2 show the needs of implementing very basic and common ACRs such as whitelist and blacklist. Note that the involved lists should be updatable dynamically, and the method for defining the lists should be flexible enough. Implementing even such a basic access control on-chain in smart contracts would be highly expensive if a black/whitelist is long and/or updated frequently. Moreover, managing such an access control list would be impractical as executing blockchain transactions is significantly delayed (minutes to hours [26]). The list maintained on-chain would be also visible to anyone which in most cases is undesired.

Example 3. *The owner of a contract may require that only authorized parties can call a specific method. She may demand more fine-tuned controls: only authorized parties can call the method with specific arguments.*

This kind of selective restrictions may be useful in almost any smart contract application, e.g., to determine who can move the money, who can stop a service, etc. The latter rule

hints at an even more exciting application of access control: *the owner can specify that an address can call a method in a smart contract only when the payload will not trigger any known security problems.* By extending this kind of rules, we may prevent attacks on vulnerable contracts even *after their deployment* – see the next example.

Example 4. *The owner of a contract may require that a given call can be executed only when it is validated by some sophisticated runtime verification tool(s).*

In contrast to the previous examples, such a rule would allow the owner to inspect a given call in detail and limit access in the case of any issue detected. In such a way the owner could benefit from runtime verification tools running them off-chain, without integrating them in the smart contract environment. The owner may also want to ensure a given call can be executed only once, unless a new permission is granted.

III. SMACS OVERVIEW

Throughout the paper, we describe SMACS in the context of Ethereum and Solidity, but it can be easily extended to other platforms and languages with similar capabilities.

A. System Participants

SMACS involves four types of actors:

- **SMACS-enabled Smart Contracts** are contracts on blockchain protected by SMACS. A SMACS-enabled smart contract verifies incoming calls by validating their corresponding tokens. Any transaction or a message call will be rejected if a valid token is not presented. For simple description, we often refer to SMACS-enabled smart contracts as “smart contracts” or just “contracts” and from the context it will be clear when we distinguish them from legacy smart contracts.
- **Owner** is the creator of a smart contract. Normally, there could be several smart contracts under the control of a single owner. An owner is responsible for defining and managing the ACRs. Also, an owner needs to manage a Token Service (TS) instance corresponding to a SMACS-enabled smart contract.
- **Token Service (TS)** is a service that is responsible for verifying requests from clients and issuing access control *tokens* accordingly. A token issued by a TS determines exact access permissions of a particular client with respect to a SMACS-enabled smart contract.
- **Clients** are users who want to access the resources (e.g., data, methods) of SMACS-enabled smart contracts. A client must obtain a token granting appropriate permissions from the TS before she can access the smart contracts.

B. Goals

We design SMACS with the following goals in mind.

- a) **Security:** We assume that an adversary cannot compromise the underlying cryptographic primitives (e.g., signatures, hash functions, etc.) and cannot compromise the runtime environment of the deployed smart contract platform. However, we discuss an adversary able to reverse the blockchain history (i.e., launch a *51% attack*). Under these assumptions SMACS

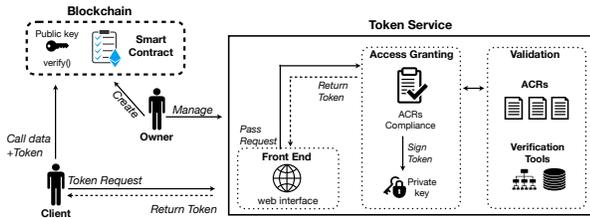


Fig. 1: Details of the SMACS framework.

should prevent unauthorized entities from accessing SMACS-enabled smart contracts. When an entity is allowed to access a SMACS-enabled smart contract, its behavior cannot deviate from the permission it has been granted. Moreover, apart from enabling access control in the traditional sense, SMACS should be able to counter certain runtime attacks even if the underlying smart contracts are vulnerable to these attacks.

b) Flexibility and Extensibility: The SMACS framework should be able to define complex and fine-grained ACRs for smart contracts while keeping smart contracts simple. SMACS should allow to manage these rules by removing, adding, or modifying them dynamically, but without updating contracts. Also, it should be easy to extend SMACS by integrating smart contract protection techniques of various classes.

c) Efficiency and Low cost: SMACS should operate as efficiently as possible. There should be no efficiency bottlenecks with respect to throughput, storage, latency, etc. which could hinder its applicability in real-world scenarios. The cost of applying SMACS in terms of storage, computation, and blockchain-related fees should be minimized. Meanwhile, the process of integrating and deploying SMACS-enabled smart contracts should be easy and intuitive, not incurring a high development effort and cost.

C. Overview

In SMACS, the owner first generates a public and private key pair (pk_{TS}, sk_{TS}) , and preloads the Token Service (TS) with sk_{TS} and an initial set of ACRs (or token issuing rules). The private key sk_{TS} will be used by the TS to sign tokens it issues, while the ACRs specify the condition under which a token can be issued. The owner is also responsible for creating the SMACS-enabled smart contract with the public key pk_{TS} preloaded. The SMACS-enabled smart contract verifies the validity of the access credentials (tokens) of the incoming calls with the public key pk_{TS} before continuing an actual execution. We use $Sign_{sk}(\cdot)$ and $SigVerify_{pk}(\cdot)$ to denote the signing with sk and verifying with pk in later sections.

Although tokens in SMACS can have different types and can be issued basing on arbitrary validation logic, SMACS-enabled smart contracts stay simple and implement only an easy access control verification. In most cases, the only overheads that SMACS introduce are a) storing a public key, b) parsing a token, and c) a signature verification per call. In our design, the burden of all memory consuming and computationally heavy operations are shifted to an off-chain TS. All intended ACRs are initialized into TS. These rules can be updated dynamically by the owner. Before accessing the SMACS-enabled smart

contract, a client must apply for a token with compatible permissions from TS. Upon receiving the request from a client, the TS checks the request against the ACRs. If the request does not violate the rules, the TS issues a token to the client by signing the datagram formed by relevant information extracted from the request and metadata. The client then constructs a transaction with the token encoded into it to access the smart contract. The transaction constructed by the client has to be compatible with her previous request, since the signature creates a cryptographic binding and any attempt to modify the actual transaction will make the signature verification fail. In practice, the situation may be more complex. A client may issue a transaction that triggers the execution of a chain of smart contracts. How to handle these situations will be clear in the following sections.

IV. SMACS DETAILS

The detailed architecture of SMACS is shown in Fig. 1. The owner and clients are external owned accounts operated via the client-side software (usually called a *wallet*) to interact with SMACS-enabled contracts. The TS consists of a) the *front end* interface, b) the *access granting* module that checks the rules compliance and issues tokens, and c) the *validation* module that contains all verification tools (if any) and respective rules. These rules determine who can get a token with particular permissions. The owner and clients interact with the TS through an HTTPS-enabled web interface provided by the TS. The realization of the access control of the smart contract is ultimately due to the control of the issuance of tokens.

A. Token Types

SMACS supports three different types of tokens with different permission semantics. These types are designed to facilitate a flexible and fine-grained access control over the SMACS-enabled smart contracts.

- **Super token** is of the highest permission level. A client with a super token can freely call all public methods of the smart contract with arbitrary arguments before the token expires.
- **Method token** limits the access to a specific method. A client with a method token can call the specific contract's method associated with the token with arbitrary arguments before the token expires. A method token issued for a particular method cannot be used to access other methods.
- **Argument token** is similar to a method token with the additional restriction that the associated method can only be called with specific arguments.

All tokens are issued with an expiration time set by the TS. The expiration time determines the token lifetime, i.e., until when the token can be used to authorize the corresponding calls. Any token can have the *one-time* property set. A one-time token gets invalidated once it is used to successfully access the smart contract, which ensures that the token holder can only access the smart contract once with issued token.

Before a client can get a token from the TS, she has to submit a well-formed *token request* to the TS. As depicted in Fig. 2, the token request varies according to the requested

Tab. I: Elements of a token request payload.

Type	reqPayload				
	cAddr	sAddr	methodId	argName	argValue
Super	✓	✓	✗	✗	✗
Method	✓	✓	✓	✗	✗
Argument	✓	✓	✓	✓	✓

token type, where `type` is a token type, `cAddr` is the address of a targeted contract, `sAddr` is the address of a client’s account, `methodId` is the method identifier that is going to be accessed with a method or argument token, `argName` and `argValue` are the argument and argument value used when an argument token is requested (there can be multiple argument-value pairs passed in a token request).

In SMACS, a token is implemented as an 86-byte object shown in Fig. 3, where `type` indicates the type of a token, `expire` encodes the expiration time, `index` is used for tokens with their one-time property set (if the value of `index` is a non-negative integer, then the one-time property is set), and the `signature` field is computed as:

$$\text{Sign}_{sk_{TS}}(\text{type} \parallel \text{expire} \parallel \text{index} \parallel \text{reqPayload}),$$

where `type` and `reqPayload` are extracted from the token request sent by the client (see Fig. 2). The `reqPayload` is an optional field of the token request with variable size according to `type`. Its exact formulation is shown in Tab. I.

B. Token Issuance and Verification

There are two verification processes in SMACS: a TS verifies incoming token requests against its rules, and a SMACS-enabled contract verifies tokens extracted from incoming transactions. Any failed attempt to access the contract is ultimately due to the failure of one of these two verification processes.

a) **Token Issuance:** To apply for a token, a client sends a token request specifying the intended `type` together with a compatible `reqPayload`, which describes who (`sAddr`) will access which (`cAddr`) smart contract and how it will be accessed. The request payload (`reqPayload`) depends on the intended `type` (see Tab. I). When receiving the token request, the TS parses and checks it against the rules. Once verified, a token is issued according to the request. This step can be easily integrated into mainstream wallets, such that it is executed seamlessly for users prior to actual transaction sending.

b) **Contract-side Verification:** Once getting a token from the TS, the client can construct a transaction whose calldata is filled with the token together with other necessary information that is compatible with the token (i.e., the token will be passed as an argument). Upon receiving the transaction, the token verification process shown in Alg. 1 is triggered, and only

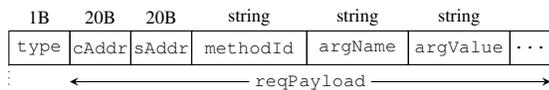


Fig. 2: The layout of a token request.

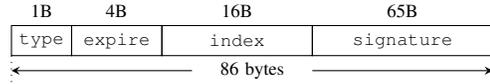


Fig. 3: The layout of a token.

after the process succeeds the smart contract can continue to execute the transaction accordingly.

The SMACS-enabled smart contract first extracts the token from the transaction, checks whether it expires, and whether it has been used if the one-time property is set. We discuss the one-time property validity checking (i.e., the details of `reused()` check) in § IV-C. Then the SMACS-enabled smart contract reconstructs the data required to verify the token signature according to the type of the token. In this step, the smart contract uses the EVM’s transaction context objects (see § II-C) to make sure that the passed ticket matches the current transaction.

We emphasize that this verification process is generic and the implementation of SMACS-enabled smart contracts respects the common development flows. Basically, the code of legacy smart contracts can be made deployment-ready in the SMACS framework by ensuring that every method callable from outside (i.e., public or external) verifies a token prior to its actual body execution. To achieve it, for each public and external method the `tokens` argument is added to the original argument list and the `verify` call performing Alg. 1 is asserted prior to the actual method body.

To facilitate easy adoption we develop a tool that allows to transform any legacy smart contract into an equivalent SMACS-enabled smart contract. An example of such a transformation is presented in Fig. 4 (note that internal methods do not have to verify tokens and in the case of public/external methods called internally they are split into separate methods).

C. One-time Tokens

The one-time property ensures that a given token can be used only once. One-time tokens may be especially useful

Alg. 1: Contract-side token verification.

Input: A transaction T
Output: The verification result

```

 $tk \leftarrow \text{extractToken}(T)$ 
if  $\text{now}() > tk.\text{expire}$  then
   $\perp$  return False
if  $tk.\text{index} > -1$  and not  $\text{reused}(tk.\text{index})$  then
   $\perp$  return False
 $tkData \leftarrow tk.\text{expire} \parallel tk.\text{index}$ 
 $addrData \leftarrow T.\text{origin} \parallel \text{address}(this)$ 
 $data \leftarrow tk.\text{type} \parallel tkData \parallel addrData$ 
if  $tk.\text{type} = \text{Super}$  then
   $| data \leftarrow data$ 
else if  $tk.\text{type} = \text{Method}$  then
   $| data \leftarrow data \parallel msg.\text{sig}$ 
else if  $tk.\text{type} = \text{Argument}$  then
   $| data \leftarrow data \parallel msg.\text{sig} \parallel msg.\text{data}$ 
return  $\text{SigVerify}_{pk_{TS}}(data, tk.\text{signature})$ 

```

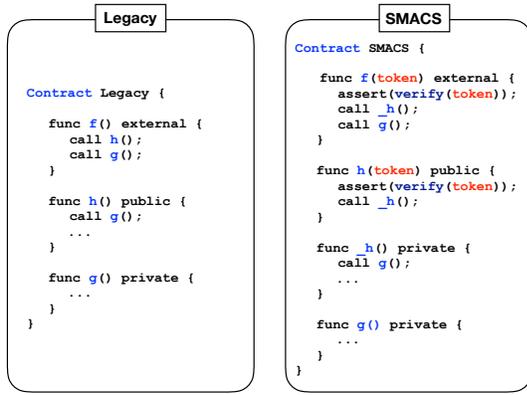


Fig. 4: Automated SMACS adoption for a legacy contract.

for access control of security-critical methods or for example when a client is unknown to the requested TS. To realize this property, one may be tempted to rely on the *nonce* mechanism employed by Ethereum for counteracting replay attacks. However, we emphasize that the transaction’s nonce cannot be accessed by the smart contract itself. Therefore, SMACS has to implement an in-contract mechanism to support the verification of one-time tokens.

The TS maintains `counter` variable (corresponding to the `index` field of a token) for issuing tokens with the one-time property set. `counter` is initialized to 0, whenever a new one-time token is being issued, it is incremented by 1, and the updated value is used as the `index` value for this token.

Since a one-time token is supposed to be used only *once*, when a client with such a token tries to access an SMACS-enabled smart contract, the contract has to check whether the underlying token has been used before, and then permits or denies the access attempt accordingly. A trivial way for the contract to realize this is to store the `index` values of all one-time tokens having made a successful access. However, as the on-chain storage is expensive, this approach can be costly and impractical if the number of issued one-time tokens is large.

Based on the observations that the TS can assign every one-time token with a unique index consecutively and the token lifetime is limited, we propose a cost-effective scheme to handle one-time tokens, where every index is efficiently encoded as a single bit of a cyclically reused bitmap. In our approach, an n -bit map S (together with its internal state) is used to represent the status (used or unused) of a set of n one-time tokens with consecutive `index` values. The state of the bitmap can be represented by a tuple $(S, start, startPtr, end, endPtr)$, where $S \in \{0, 1\}^n$, $start \in \{0, 1, \dots\}$, $startPtr \in \{0, \dots, n-1\}$, $end = start + n - 1$, and $endPtr = startPtr + n - 1 \bmod n$. In Alg. 2, the n -bit sequence $S[startPtr] \parallel S[startPtr + 1 \bmod n] \parallel \dots \parallel S[endPtr]$ indicates the status of the n one-time tokens with indexes $start, start+1, \dots$, and end . A token with the index i is regarded as unused if and only if a) $i \in \{start, start+1, \dots, start+n-1\}$ and $S[(startPtr+i-start) \bmod n] = 0$, or b) $i > end$. When the index i of a token is unused, the state of the bitmap is updated according to Alg. 2.

Alg. 2: Bitmap state update.

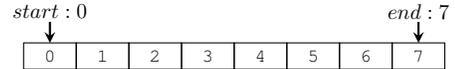
```

/* Initialization */
S ← [0, …, 0]; start ← 0; end ← n - 1;
startPtr ← 0; endPtr ← n - 1;
/* Update */
i ← getIndex(token)
if i < start then
  return False
else if start ≤ i ≤ end then
  t ← (startPtr + i - start) mod n
  if S[t] = 1 then
    return False
  else
    S[t] ← 1
    return True
else if end < i ≤ end + n then
  startPtr ← seek(S, i, end, startPtr)
  endPtr ← (startPtr + n - 1) mod n
  end ← i; start ← end - n + 1; S[endPtr] ← 1;
  return True
else
  /* Reset when i is too large */
  S ← [0, …, 0]; startPtr ← 0; endPtr ← n - 1;
  start ← i; end ← i + n - 1; return True

/* seek(S, i, end, startPtr) returns the smallest
integer j in {0, …, n-1} such that S[j] = 0
and i - end ≤ j - startPtr */

```

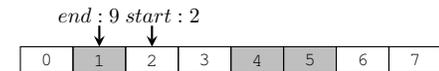
Let us consider a bitmap S with size $n = 8$. At the beginning, all cells of S are set to 0 with $start = startPtr = 0$ and $end = endPtr = 7$.



After tokens whose `index` are 0, 1, 4, and 5 access the smart contract, the corresponding cells are set to 1 (gray cells).



Upon receiving the access request by the token with `index` 9, `seek()` is responsible for finding the updated `startPtr`. Since $end = 7 < 9 \leq end + 8 = 15$, `seek()` returns 2, which is assigned to `startPtr`, and $S[endPtr]$ is set to 1, where $endPtr \leftarrow startPtr + n - 1 \bmod n = 2 + 8 - 1 \bmod 8 = 1$.



At this point, $S[2] \parallel S[3] \parallel \dots \parallel S[7] \parallel S[0] \parallel S[1]$ represents the status of the tokens with indexes in $\{2, 3, \dots, 9\}$. We continue to consider a more complicated case where an access request is made by the token with `index` 13. Then the state of the bitmap is updated as follows.



This state only represents the status of the tokens with indexes in $\{6, \dots, 13\}$, and the information of the unused tokens with

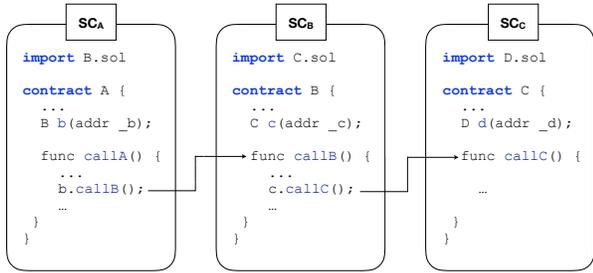


Fig. 5: A call chain with the depth three.

indexes 2 and 3 is lost (access requests originated from these two tokens will be rejected). Moreover, if an access request is made by a token with very large index i such that $i > end + n$, the bitmap will reset all cells to zero and update the pointers according to Alg. 2.

Therefore, the bitmap approach ensures that any one-time token can be used at most once. It is possible that in certain situations some one-time tokens become invalid before they are used, which is called a token miss. For example, if the smart contract has processed a token with the index 13, the range of the bitmap is updated to $start=6$ and $end=13$. This implies that any (even unused) token with an index smaller than 6 will be rejected (missed) by the contract. In this case, a holder of such an unused token would need to re-apply for a new token from the TS again. To avoid this situation, an owner should allocate a large-enough bitmap in its smart contract. There is a trade-off between the size of the bitmap and the miss rate. The two factors that allow to model the bitmap size are a) a token lifetime, and b) the (expected) maximum number of transactions per second that the contract is going to process. Then the bitmap size required to not reject any unused and non-expired token is $token_lifetime \times max_tx_per_second$ bits. Fortunately, as we show in § VI-A, even for the most popular Ethereum contracts and realistic token lifetimes, the cost of the bitmap storage is low.

D. Tokens for Call Chains

Starting from a transaction originated by an external owned account, an invoked contract method can call a method of another contract which in turn can call a method of a third contract, and this call chain (see § II-C) can go on.

When the smart contracts involved in the call chain are protected with SMACS, the client initiating the call chain has to obtain proper tokens for all these smart contracts. Let us consider a simple example shown in Fig. 5. Before a client triggers this call chain, she needs to obtain three tokens (e.g., method tokens) from the TSes corresponding to SC_A , SC_B , and SC_C (these TSes can be operated by different owners).

Assuming that the client successfully gets the three tokens tk_A , tk_B , and tk_C , then she can embed an array of the three tokens of the following form in the transaction:

$$SC_A : tk_A \parallel SC_B : tk_B \parallel SC_C : tk_C.$$

```

1  {
2    sender: {
3      whitelist: ["0x366c...", "0xd488...", "..."],
4    },
5    method: {
6      methodA: {
7        blacklist: ["0xba7F...", "0xb1D4...", "..."],
8      },
9      ...
10   },
11   argument: {
12     argA: {
13       whitelist: ["0x3540...", "0x9e9B...", "..."],
14     },
15     ...
16   }
17 }

```

Fig. 6: An example of whitelists and blacklists.

When SC_A receives the transaction, it can extract the token (tk_A) associated with its address and verify validity according to Alg. 1. Subsequently, this array of tokens will be passed to SC_B via a message call who parses out tk_B and verifies it. Finally, the array is passed while calling SC_C which can perform the analogous operations as SC_A and SC_B .

E. Access Control Rules (ACRs)

Rules in SMACS define the set of token requests which can successfully get the token from a TS when submitted. For every token type, there is a set of rules associated with it. A token request of a particular type will be checked against the set of rules associated with that type. In the following section, we present sample rules that can be implemented in SMACS.

a) **Blacklist and Whitelist:** Blacklist and whitelist are generic ACRs supported in SMACS. In our context, the simplest form of a whitelist is just a set of Ethereum addresses. Every address outside this list cannot obtain a valid token from the TS, and therefore it cannot access the SMACS-enabled smart contract. As depicted in Fig. 6, each token type has either a blacklist or whitelist. SMACS does not mandate how these lists are created and for instance an address whitelisted for super tokens can be blacklisted for argument token. Moreover, the listed objects are not necessarily account addresses. For example, it is possible to blacklist dangerous argument values for certain contract methods. In SMACS, all these access lists can be dynamically updated by the owner without any modification to the deployed smart contract.

b) **Rules for Runtime Verification:** Apart from these basic rules above, the argument token type allows us to craft more advanced ACRs that can enhance the runtime security of the SMACS-enabled smart contract. Imagining that a client tries to access a method of a smart contract with a particular set of arguments. SMACS allows TSes to simulate the runtime behavior of the smart contract in an isolated off-chain environment and deny access if any abnormal behaviors triggered by the requested call are observed. Then a TS implementing proper rules for argument tokens would be able to protect even vulnerable already deployed smart contracts. We show concrete instantiations of such rules in the next section.

V. RUNTIME VERIFICATION CASE STUDIES

Defensive logics with arbitrary complexity can be plugged into SMACS. In particular, SMACS can be powerful when combined with any other runtime verification tools preventing specific attack classes. In this section, we show two concrete examples where third-party tools are employed to implement advanced rules enforcing certain runtime security properties.

A. Enforcing Hydra Uniformity

Hydra is a recent framework for smart contract bug bounty administration, which enables runtime detection and rewarding of critical bugs [27]. Basically, in the Hydra framework, multiple independent program instances written in different programming languages but with the same intended high-level logic run in parallel over the blockchain. These program instances are called the heads (of the Hydra).

When a smart contract protected by Hydra executes, its intended logic proceeds normally only if the outputs of the heads are identical. If the outputs diverge for different heads, it is likely that certain erroneous state is triggered for some heads. At this point, the execution of the smart contract aborts and the rewarding logic of Hydra takes control to pay out a bounty. Therefore, Hydra can detect bugs at runtime at the cost of increased on-chain resource consumption by a factor round N when N heads are employed.

We integrated Hydra into SMACS by defining a dedicated rule for argument tokens. This rule dictates that an argument token is issued only when the outputs of all heads are identical when called with the payload specified in the token request. In contrast to Hydra, heads in SMACS are run by a TS on its local testnet. Hydra acts as a simulator in SMACS, does not consume on-chain resources, and therefore it is possible to implement more heads in our case without introducing additional on-chain cost. In summary, this rule enforces Hydra uniformity, where transactions leading to different head outputs are unable to get a token at the first place. We show the efficiency of Hydra-supported SMACS in § VI-B.

B. Blocking Re-entrancy Attacks

In this case, we show how to protect a smart contract from the so-called *re-entrancy attack*, the essence of the real-world TheDAO attack [12], leading to a loss of over \$50 million worth of Ether at the time the attack occurred.

Let us consider the vulnerable smart contract `Bank`, a simplified version of TheDAO [28], as shown in Fig. 7. Anyone can deposit ether into `Bank`, the amount of ether is recorded in the mapping `balance`. The ether deposited can be withdrawn by calling `withdraw()`, which sends the ether to the `msg.sender` address. This transfer implicitly triggers a *fallback method* (an anonymous method that does not take any arguments) of the receiver. This default behavior can have security consequences as the execution flow can be controlled by a remote fallback method. The re-entrancy attack can be lunched by an attacker using the smart contract shown in Fig. 7. She first calls the `deposit()` method to deposit two ethers into the target smart contract `Bank`. Now she is

```
1 contract Bank{
2   mapping(address=>uint) balance;
3   function addBalance() public{
4     balance[msg.sender] += msg.value;
5   }
6   function withdraw() public{
7     uint amount = balance[msg.sender];
8     if (msg.sender.call.value(amount)() == false)
9       {throw;}
10    balance[msg.sender] = 0;
11  }
12 }
13 contract Attacker{
14   bool isAttack; address bank;
15   function Attacker(addr _bank, bool _isAttack){
16     bank = _bank; isAttack = _isAttack;
17   }
18   function() payable{
19     if(isAttack == true){
20       isAttack = false;
21       if(bank.withdraw()) {throw;}
22     }
23   }
24   function deposit(){
25     bank.call.value(2).addBalance();
26   }
27   function withdraw(){
28     bank.withdraw();
29   }
30 }
```

Fig. 7: The `Bank` contract with a re-entrancy vulnerability and the `Attacker` contract exploiting it.

ready to attack the target by calling the `withdraw()` method of `Attacker`. Subsequently, `Attacker.withdraw()` calls `Bank.withdraw()` which then triggers a recursive `Bank.withdraw()` call via `Attacker`'s fallback method, and the line 11 of the `Bank` smart contract is never reached. The above attack strategy effectively moves all ether from `Bank` to the account controlled by the attacker.

To prevent `Bank` from being exploited, we use SMACS with a rule employing `ECFChecker` [29] – a developed tool for detection of *effectively callback free* objects [12]. To integrate that, the TS deploys an `ECFChecker`-supported implementation running an off-chain testnet with the `Bank` contract deployed. For every token request, the TS calls a requested method with the passed arguments and observes the output of `ECFChecker`. The TS issues the tokens only if `ECFChecker` does not report any security issue. We emphasize that the described integration gives the contract owner `ECFChecker` security benefits without requiring Ethereum participants to update their configurations to support `ECFChecker`. In § VI-B we show the efficiency of this setup.

VI. IMPLEMENTATION AND EVALUATION

To evaluate our design, we fully implement the SMACS framework. SMACS-enabled smart contracts are developed by `Solidity v0.4.24` and deployed on a testnet. The TS is implemented as a web server running `Node.js v10.2.1` bundled with the `node-localStorage` package for storing rules and signature key-pairs. We implement client and owner with `web3.js` [30]. This software interacts with deployed SMACS-enabled smart contracts and TSes. We use

Tab. II: Single token processing gas cost.

Cost	Token type (without the one-time property)		
	Super	Method	Argument
Verify	108282 (65%)	115108 (67%)	330889 (85%)
Misc	57675 (35%)	57675 (33%)	57678 (15%)
Total	165957	172783	388567
USD	0.041	0.042	0.094

Cost	Token type (with the one-time property)		
	Super	Method	Argument
Verify	108531 (56%)	115651 (58%)	330914 (79%)
Misc	57426 (30%)	56994 (28%)	57331 (14%)
Bitmap	27471 (14%)	27839 (14%)	28003 (7%)
Total	193428	200484	416248
USD	0.047	0.048	0.101

Tab. III: Gas cost for multiple one-time argument tokens.

Cost	Number of Token			
	1	2	3	4
Verify	330914 (79%)	662952 (79%)	994552 (78%)	1326506 (78%)
Misc	57331 (14%)	102991 (12%)	150463 (12%)	203499 (12%)
Bitmap	28003 (7%)	56746 (7%)	84612 (7%)	112034 (7%)
Parse	–	16986 (2%)	34182 (3%)	57872 (3%)
Total	416248	839675	1263809	1699911
USD	0.101	0.204	0.307	0.412

the Ethereum’s ECDSA signature scheme as the default one, as Ethereum provides a native and optimized support for it.

A. Gas Cost

In the SMACS framework, clients send transactions with proper tokens which are verified by smart contracts. Therefore, the main cost is introduced with respect to the computation and storage whose utilization is charged by the Ethereum network. We perform a series of experiments to measure the cost introduced by SMACS in terms of gas consumption.

We conduct experiments for different types of tokens and record their gas cost, together with the cost converted to US dollars in Tab. II. The conversion was according to the gas price from [31] at the time of writing the paper. From the table, we can see that the dominating operation is the signature verification. The cost also increases in arguments tokens as they require more processing (`argName` and `argValue` have to be processed). However, the overall cost of a token verification is around \$0.04 for super and method tokens and around \$0.1 for argument tokens. As shown in the table, for tokens with the one-time property the verification gas consumption is similar, despite additional operations are required by the bitmap.

As discussed in § IV-D, SMACS supports transactions that invoke a call chain of contracts. In this case, the token verification cost varies according to the depth of the chain, and additional cost is induced since a contract has to parse the passed token array before verification. We conduct analogical experiments as in the previous case and the results are shown in Tab. III and Fig. 8. (Note that the table presents the results for the argument token type whose verification is around two times more gas consuming than other types.) As presented, the verification cost increases linearly with the call chain length.

Tab. IV: Storage cost for the bitmap (this cost is one-time).

Cost	Transaction frequency (tx/s)		
	35	3.5	0.35
Storage	15.38 KB	1.54 KB	0.154 KB
Deployment	8849037	886054	88605
USD	2.140	0.214	0.021

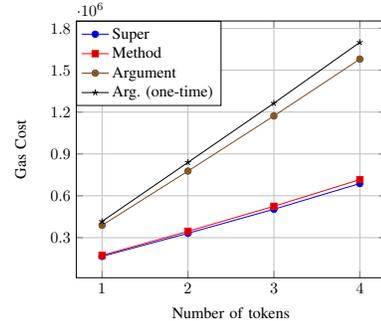


Fig. 8: Aggregated gas cost for verifying multiple tokens.

Implementing the one-time property requires to store a bitmap by smart contracts. The size of this storage depends on the token lifetime and the expected transaction frequency, however, this cost is one-time, paid upon the contract creation. To give insights on the cost we take the ten most popular smart contracts based on the number of transactions by Jan, 2019 [32] and analyze their transactions distribution. We found that on average the transaction peak is 35 *tx/s* which is close to the Ethereum’s maximum throughput [33]. Setting the lifetime of one-time tokens to one hour and assuming conservatively that all transactions use one-time tokens, Tab. IV shows the required storage and its cost. We can see that to handle even 35 *one-time tokens/s* a smart contract has to be initialized with storage costing only one-time fee of \$2.14. This cost is linear in transaction frequency and token lifetime.

B. Token Service Performance

a) **TS Throughput:** We evaluate the TS throughput running a TS instance on a system with macOS Sierra 10.12.6, Intel Core i5 CPU (2.7 GHz), and 8GB RAM. For each token type, we send 10^i ($0 \leq i \leq 5$) token requests to the TS, record the total time needed by the TS, and compute the average time required per token request. The rules used are composed of blacklists and whitelists as presented in Fig. 6. The obtained results are summarized in Fig. 9.

From Fig. 9 we can see the number of token requests handled per second raises when the requests are processed in batches. The throughput becomes stable when the number of requests is greater than 10^5 , with the time cost about 5ms for most token types. The single TS instance can easily handle all transactions processed by the current Ethereum main network even in peak times. We found the ever highest transactions peak in Ethereum for one of the most popular smart contracts – CryptoKitties [34] when it received about 48 transactions per second (on 05-Dec-17 00:43:03 UTC [35], [36]).

b) **Integration with Runtime Tools:** In § V we integrate SMACS with two runtime verification tools, i.e., Hydra and

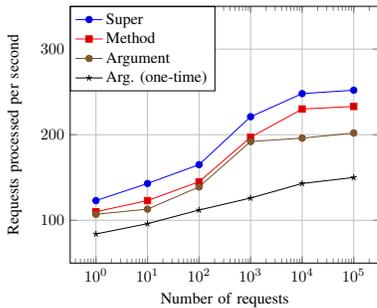


Fig. 9: Throughput of the TS.

ECFChecker. In both cases, TS to verify incoming requests requires *local* Ethereum testnets (no code changes at the deployed contracts or Ethereum software used by network nodes are required, however). To improve throughput of the tools we configure the testnets (i.e., geth [37]) to minimize the latency between submitting and executing transactions. For Hydra, we implement a simple contract in three different programming languages and deploy it on a Hydra-supported testnet. For ECFChecker, we deploy the vulnerable contract presented in § V. We send 100 transactions each and measure the average time needed by a tool to process a transaction. In our setting, SMACS with Hydra needs 120ms to process a request, while ECFChecker-supported SMACS can process a request in only 10ms. Thus, with these tools SMACS can handle around 8 and 100 token requests per second.

VII. DISCUSSION

A. Security

Our first claim is that an adversary cannot bypass the access control in SMACS. All contract’s public interfaces require the token verification process (see § IV-B). This process ensures that a token is authentic (i.e., signed by the TS), non-expired, and matches the calling transaction. The only way of obtaining such a token is to request the TS which would verify the request against its access rules. All valid tokens are signed by the TS, therefore an adversary without passing the TS validation or without the TS’ private key cannot get a valid token for its transactions. Moreover, one-time tokens could be issued only the clients satisfied the rules predefined in whitelist, guaranteeing one-time access even if an adversary controls multiple addresses.

a) Substitution Attack: An attacker can intercept a transaction from a legitimate client, extract the token from it, and then construct a transaction with the intercepted token. This transaction will be rejected by the contract-side verification, since the signature of the TS creates a cryptographic binding of the token and the context in which the token can be applied. Any tiny change of the context (e.g., address, argument, etc.) will be caught by the signature verification process.

b) Replay Attack: An attacker can capture the transaction originated from a legitimate client, and replay it in the Ethereum network. This attack is against Ethereum itself and cannot succeed since the built-in countermeasure of Ethereum against replay attack will reject the replayed transaction. The

nonce value included in the transaction ensures that every single transaction is unique. If a transaction has been accepted by Ethereum, it will not be processed again. Moreover, the client’s address is protected by a token’s signature, thus, the attacker cannot extract and reuse others’ tokens. SMACS implements in-contract replay protection for one-time tokens by assigning each such a token with a unique index (set by the TS) and recording every use of a one-time token in the stored bitmap. A client can try to replay a one-time token by creating a new transaction with the used token embedded. Such a transaction will not trigger an actual execution of the targeted method, as the token verification procedure (see § IV-C) will check whether the token index was already used, and in that case deny access.

c) 51% Attack: In the 51%-attack an adversary possesses more than 50% of the total voting power of the blockchain network, what allows her to rewrite the blockchain history. This kind of attacks is devastating as they allow to double-spend, however, in our context even such a strong adversary cannot bypass the SMACS access control. The adversary can disorder or even remove transactions at will, compromising the availability of smart contracts, but she cannot obtain a valid token for a non-compliant transaction.

d) Privacy: SMACS moves access rules to TSeS which are off-chain services. Therefore deployed rules, verification tools, and their configurations are kept private and are not revealed even to clients. As blockchain transactions are publicly visible, an adversary can learn successful access control cases and try to predict the applied rules, however it is still a black-box analysis (in contrast to any in-contract access control).

B. Deployment

a) Availability: Requiring a TS to keep verifying and signing tokens introduces a single point of failure, as with the failed TS clients would not be able to interact with the contract. Fortunately, TSeS in SMACS are easy to scale and replicate. For issuing tokens (without the one-time property) providing availability is as easy as providing redundant TSeS that do not require any coordination (except a load-balancer/failover system). If a TS service is offering one-time tokens, then its replicas have to coordinate on the current counter value (see § IV-C). That can be efficiently realized via a replicated counter primitive usually implement upon a standard consensus algorithm [38], [39].

b) Service Discovery: We implicitly assumed that clients know how to reach the TS corresponding to a SMACS-enabled smart contract. In practice, clients have to learn an URL address of the service. We propose to implement this discovery process by adding the service address as a smart contract instance metadata (similarly as contract’s name or the compiler version it was created with).

VIII. RELATED WORK

In practice, the community has developed some design patterns and even third-party libraries to facilitate the application of access control over smart contracts [25]. However, this

paradigm puts the burden of all access control logic and its management on the smart contract itself. Due to the high cost of on-chain resources, only simple and inflexible ACRs can be developed using this approach (e.g., a blacklist or whitelist with small size, a role-based ACRs supporting a very limited number of roles, etc.). In summary, although smart contracts access control is an obvious need and an important aspect of smart contract security which has been extensively investigated over the last years, we are not aware of any framework similar to SMACS, which could implement complex ACRs supporting runtime security verification at a very low cost. The most relevant research to SMACS is the investigation and development of methodologies and tools for detecting vulnerabilities of smart contracts, which can be divided into two general categories: *static* and *runtime* security analysis.

a) Static Security Analysis: These methods or tools mainly based on formal verification and symbolic execution. Oyente [8] and Manticore [9] are symbolic execution tools for finding potential security bugs. Mythril [10] uses concolic analysis, taint analysis, and control flow checking to detect multiple smart contract security vulnerabilities. Securify [11] extracts semantic facts by performing advanced static analysis to prove the presence or absence of certain security vulnerabilities. Zeus [40] employs model checking to verify the correctness of smart contracts. MAIAN [41], processes the bytecode of smart contracts and tries to build a trace of transactions to find and confirm bugs based on inter-procedural symbolic analysis. The list of tools are difficult to enumerate and new relevant tools are constantly emerging [20], [42], [43], [44]. Most of these tools are meant to provide pre-deployment security verification. Thus they can only identify bugs (rather than protect from them) for already deployed smart contracts.

Another drawback is that it cannot fully cover all runtime behaviors and therefore is susceptible to missing novel runtime attack patterns. In fact, this has been demonstrated in [13], where new re-entrancy attack vectors are crafted which bypass the security check of existing static analysis tools [40], [8].

We see this class of tools as orthogonal to SMACS, however, we believe that in some cases they could be used in combination providing security benefits. For example, the owner of a SMACS-enabled smart contract can scan the deployed contract regularly with such tools (e.g., perform a vulnerability scan whenever the security analysis tools get updated). Once a vulnerability is detected, she can blacklist transactions with specific patterns that can potentially trigger the vulnerability.

b) Runtime Security Analysis: In contrast to static security analysis, tools [12], [14], [13] performing runtime monitoring has the potential to prevent deployed smart contracts from being exploited. Hydra [14] enables post-deployment security through *N-of-N-version* programming, a variant of classical *N-version* programming that runs multiple independent program instances to detect runtime security issues. ECFChecker [12] is a runtime detection tool dedicated to finding *effectively callback free* objects. This tool can be used for finding Ethereum re-entrancy attacks. More detailed overview of Hydra and ECFChecker can be found in § V.

Another interesting example is the Sereum [13] architecture, a hardened EVM which is able to protect deployed contracts against re-entrancy attacks in a backward compatible way by leveraging taint tracking to monitor runtime behaviors of smart contracts. Sereum can also be integrated into the SMACS framework easily by using dedicated ACRs.

The main drawback of these tools is their requirement of changing and upgrading the runtime environment. We emphasize that in the replicate state machine model followed by blockchain platforms, this implies that a majority of nodes would need to update their EVMs to support such a tool. SMACS enables contract owners to benefit from these tools without this requirement. Moreover, as we presented, these tools can be easily and seamlessly integrated with SMACS. Another preferable feature offered by combining runtime security analysis tools with SMACS is that a vulnerable smart contract may still operate normally, since only innocent transactions pass through and suspicious transactions identified by the tools are rejected at runtime.

IX. CONCLUSIONS

We presented SMACS, to the best of our knowledge, the first framework that achieves efficient, flexible, and fine-grained access control of smart contracts with low cost by combining lightweight on-chain verifications and off-chain access control management infrastructures. Apart from enabling malicious addresses prevention and abnormal runtime behaviors resistance for smart contracts, SMACS offers several preferable features. Firstly, when combined with runtime verification tools, a SMACS-enabled smart contract can deny suspicious access attempts on the fly while keeping operating for innocent transactions. Secondly, the architecture of SMACS allows rules for enhancing *post-deployment* security to be designed based on which it is possible to prevent vulnerabilities discovered *after deployment* from being exploited. Therefore, it is meaningful to test SMACS-enabled contracts with new verification tools regularly and adjust the rules accordingly. Finally, due to the extensibility of the framework, we could expect more security-related tools that can be applied in SMACS to emerge in the future.

An interesting research direction is to investigate trusted execution environments (TEEs, e.g., Intel SGX) in the context of SMACS to fully decentralize it. For instance, a TS implemented within a TEE enclave could decentralize the entire system: an owner would just publish its ACRs which would be validated by the enclave code running locally on a client (without contacting any central service). We leave a detailed design of such a system as future work.

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd Yinzhi Cao for their valuable comments and suggestions. This research is supported by the Ministry of Education, Singapore, under its MOE AcRF Tier 2 grant (MOE2018-T2-1-111) and by the SUTD SRG ISTD 2017 128 grant.

REFERENCES

- [1] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2013.
- [2] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought*, 1996.
- [3] "Rubixi bug," <https://bit.ly/2VifC3z>, 2016.
- [4] "Hackergold token bug," <https://bit.ly/2U9JQt1>, 2017.
- [5] "Dao exploit example," <https://bit.ly/2S3Y5cE>, 2016.
- [6] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," 2017.
- [7] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.
- [8] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [9] "Manticore symbolic execution tool." <https://github.com/trailofbits/manticore>, 2018.
- [10] "Mythril tool." <https://github.com/ConsenSys/mythril>, 2018.
- [11] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *The 25th ACM Conference on Computer and Communications Security*, 2018.
- [12] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," in *44th Proceedings of the ACM on Programming Languages*, 2017.
- [13] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *26th Network and Distributed System Security Symposium*, 2019.
- [14] L. Breidenbach, I. Cornell Tech, P. Daian, F. Tramer, and A. Juels, "Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts," in *27th USENIX Security Symposium*, 2018.
- [15] S. Underwood, "Blockchain beyond bitcoin," *Communications of the ACM*, 2016.
- [16] K. Korpela, J. Hallikas, and T. Dahlberg, "Digital supply chain transformation toward blockchain integration," in *Proceedings of the 50th Hawaii international conference on system sciences*, 2017.
- [17] Y. Hirai, "Formal verification of deed contract in ethereum name service," 2016.
- [18] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 2018.
- [19] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016.
- [20] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *International Conference on Computer Aided Verification*. Springer, 2018.
- [21] D. Mohanty, "Basic solidity programming," in *Ethereum for Architects and Developers*. Springer, 2018.
- [22] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *The 23rd ACM Conference on Computer and Communications Security*, 2016.
- [23] "Globally available variables in solidity," <https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html>, 2018.
- [24] "The whitelist cost in bluzelle," <https://bit.ly/30AY1oI>, 2018.
- [25] "Openzeppelin," <https://openzeppelin.com/>, 2019.
- [26] I. Weber, V. Gramoli, A. Ponomarev, M. Staples, R. Holz, A. B. Tran, and P. Rimba, "On availability for blockchain-based systems," in *IEEE 36th Reliable Distributed Systems Symposium*, 2017.
- [27] "Hydra tool," <https://github.com/IC3Hydra/Hydra>, 2018.
- [28] "Dao exploit discovery," <https://bit.ly/2X4y0y7>, 2016.
- [29] "Ecfchecker tool," <https://github.com/shellygr/ECFChecker>, 2018.
- [30] "web3.js," <https://web3js.readthedocs.io/en/1.0/>.
- [31] "Eth gas station," <https://ethgasstation.info/>, 2018.
- [32] "Top contracts by number of transactions by 2019," https://blockspur.com/ethereum_contracts/transactions, 2019.
- [33] "Ethereum maximum throughput," <https://bit.ly/2H7wrY1>, 2018.
- [34] "Address of cryptokitties smart contract," <https://bit.ly/33SQeFH>.
- [35] "Blockspur," <https://bit.ly/2TOvOZu>, 2018.
- [36] "Etherscan," <https://etherscan.io/>, 2018.
- [37] "Geth: Official golang implementation of the ethereum protocol." <https://github.com/ethereum/go-ethereum>.
- [38] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.
- [39] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference*, 2014.
- [40] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *25th Network and Distributed System Security Symposium*, 2018.
- [41] "Maian: a tool for automatic detection of buggy ethereum smart contracts," <https://github.com/MAIAN-tool/MAIAN>, 2018.
- [42] S. Ducasse, H. Rocha, S. Bragagnolo, M. Denker, and C. Francomme, "Smartanvil: Open-source tool suite for smart contract analysis," 2019.
- [43] "Solgraph tool." <https://github.com/raineorshine/solgraph>, 2018.
- [44] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachslar-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *the 41st IEEE Symposium on Security and Privacy*, 2020.