Towards a Management Plane for Smart Contracts: Ethereum Case Study

Nida Khan *, Abdelkader Lahmadi †, Jérôme François † and Radu State *
*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
Email: {nida.khan, radu.state}@uni.lu
†Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
Email: {abdelkader.lahmadi, jerome.francois}@inria.fr

***Abstract*—Blockchain is an emerging foundational technology with the potential to create a novel economic and social system. The complexity of the technology poses many challenges and foremost amongst these are monitoring and management of blockchain-based decentralized applications. In this paper, we design, implement and evaluate a novel system to enable management operations in smart contracts. A key aspect of our system is that it facilitates the integration of these operations through dedicated 'managing' smart contracts to provide data filtering as per the role of the smart contract-based application user. We evaluate the overhead costs of such data filtering operations after post-deployment analyses of five categories of smart contracts on the Ethereum public testnet, Rinkeby. We also build a monitoring tool to display public blockchain data using a dashboard coupled with a notification mechanism of any changes in private data to the administrator of the monitored decentralized application.**

I. Introduction

Blockchain is a peer-to-peer network hosting a decentralized distributed database that provides an immutable public ledger of transactions. The primary attribute that has resulted in the growing popularity and potential prediction of complete overhauling of the financial ecosystem through its employment is the dissolution of the need of an intermediary to transfer assets. The efficacy of the technology transcends beyond the financial sector heralding the dawn of a new social and economic order [1]. Ethereum's first live blockchain network was launched in 2015 [2]. It's cryptocurrency (**Ether**) ranks third in popularity [3] after Bitcoin and Ripple. Ethereum differs from Bitcoin in being a blockchain platform on which decentralized applications can be built. It offers the service of smart contracts, which are autonomous agents that can replicate the function of legal contracts amongst other advantages [4]. Smart contracts are heralded as the most important application of blockchain. Our analysis of a dataset of 811 Ethereum smart contracts [5] shows that 80% of them have limited code complexity and very low manageability functions. Our analysis of already deployed smart contracts revealed that they had very few to none authorization and manageability functions. Smart contracts form the base of decentralized applications, which might function as products or services for mass usage. IBM is noted to have first used the acronym **RAS** (Reliability, Availability and Serviceability) to describe the robustness of its products. **M** was recently added to make the acronym **RASM** to signify the essential role **manageability** plays in supporting system robustness by facilitating many dimensions of reliability, availability and serviceability [6].

978-1-5386-3416-5/18/$31.00 © 2018 IEEE

In this paper, we provide a novel system to resolve two critical manageability issues present in the use of smart contracts in Ethereum and decentralized applications, Dapps. These are data-filtering and monitoring. To the best of our knowledge this work is the first attempt to design, develop, implement and evaluate a working management plane for smart contracts. The implemented data management templates and their associated monitoring tool can be customized and used with any smart contract or Dapp in Ethereum. We mainly developed a method, associated with monitoring operations, where access of data as per role can be changed dynamically through another managing smart contract (henceforth referred to as **MSC**). The employed design is for the monitoring and data-filtering of one smart contract (henceforth referred to as **PSC** for *Primary Smart Contract*) but can be extended to multiple PSC's being managed and monitored by one MSC. The monitoring tool and the templates have been tested on multiple smart contracts from different categories on the Ethereum public testnet, Rinkeby to evaluate the cost of implementing data-filtering and the latency in monitoring.

The remainder of this paper is organized as follows. Section II presents the motivation and the problem statement of this work. In section III, we detail the design overview of the proposed management plane. The implementation of the management plane which comprises of solidity code for a MSC template and a shell script for sending message calls to blockchain as a part of monitoring script have been elucidated in IV. Results of the experimental evaluation of data-filtering and the developed monitoring tool have been analyzed in section V. Discussion on the results is done in section VI. Related work is highlighted in section VII. Conclusion and future work are drawn in section VIII.

II. Motivation and Problem Statement

A smart contract is a program that resides and runs on the blockchain where its correct execution is enforced by a consensus protocol. It relies on a programming language provided by the blockchain platform to encode its operations and the ways to handle user transactions. It can implement a wide range of applications including gaming, financial, notary

or computation [5]. The main platform for implementing smart contracts is Ethereum and its high level programming language Solidity [7] which is compiled into bytecode language. Each compiled smart contract is stored in the blockchain and executed by the Ethereum virtual machine (EVM) running on the network nodes. Each operation in the EVM consumes **gas**. Gas is the metering unit for the use of Ethereum and helps in the calculation of fees in Ether per operation. A Solidity based smart contract is composed of a set of functions which are invoked by the contract users by sending transactions to the blockchain to validate their effects by the network. To better understand how Solidity contracts are written, we have analyzed the dataset used by Bartoletti et al [5] where they identified 811 contracts available on the blockchain explorer (*http://etherscan.io*). We focus on the distribution of number of functions per contract and how many contracts applied authorization to restrict the execution of a function according to the caller address. When analyzing the authorization in each contract, we only counted the number of functions having a solidity modifier matching the regular expression: *m*odifier $[o|O]nly$.

Figure 1 depicts the empirical cumulative distribution function (ECDF) of the number of functions per smart contract in the overall dataset and also by category. We observe that

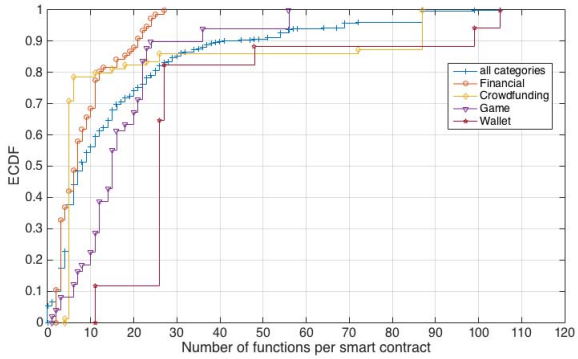

Figure 1: Distribution of number of functions per smart contract.

around 80% of smart contracts have a number of functions less than 30. In financial and crowdfunding categories this number is around 10 functions. Smart contracts are thus less complex and their number of functions is lower compared to traditional applications.

Figure 2 depicts the empirical cumulative distribution function (ECDF) of the number of filtering functions per smart contract in the overall dataset and also by category.

Overall, the number of authorization functions per smart contract is low. The number is lower than 2 for 80% of contracts. We mainly observe that already deployed smart contracts have a low manageability and authorization functions. This analysis and observations motivate our work to provide a management plane able to enforce access control and monitoring operations in smart contracts with an efficient cost.

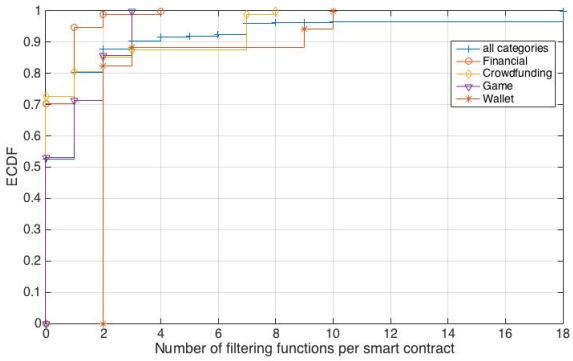

Figure 2: Distribution of number of authorization functions per smart contract.

III. Design Overview of the Management Plane

Our goal is to design a management plane to facilitate data-filtering and monitoring services to both blockchain-based applications employing smart contracts and for direct usage of smart contracts. We apply our design, implementation and evaluation of the management plane on the Ethereum platform due its popularity with an increasing number of deployed smart contracts compared to other blockchain platforms. Figure 3 depicts an overview of the elements of the architecture used to implement our management plane and its instantiation within the Ethereum platform.

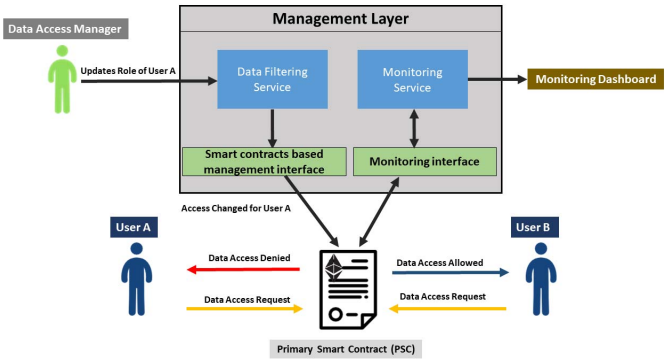

Figure 3: Functional Architecture of the Management Plane.

The management plane relies on two components namely a **data-filtering service** and a **monitoring service**. The **data access manager** interacts with the data-filtering service to update the access of a certain user to the data manipulated in the PSC. This is accomplished through a management interface relying on smart contracts, denoted as MSC's, to make the update operations. The management interface maintains the connection with the PSC whose data access control is being managed. The monitoring service collects and displays at periodic intervals the monitored data through a dashboard to the users. This is facilitated through a monitoring interface by way of which interaction with the PSC that is being monitored takes place. Both the data-filtering and monitoring service can

be implemented to work for single and multiple PSC's as shown in Figure 4.

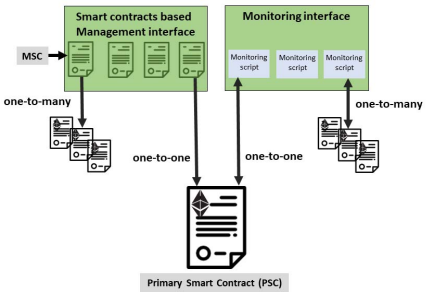

Figure 4: Interaction relationships, one-to-one and one-to-many, between management services and deployed smart contracts.

In a one-to-one relationship as depicted in Figure 4 the smart contract-based management interface has dedicated MSC's to manage data access in individual PSC's. Each MSC manages one PSC. Similarly the monitoring interface has one monitoring script to monitor data in a PSC.

In a one-to-many relationship the monitoring interface has a single monitoring script to support multiple PSC's as shown in Fig. 4. In such a relationship, the smart contract based managing interface has a single MSC managing data access in multiple PSC's.

A PSC without a MSC can be extended to include management operations but it would need to be managed separately from other PSC's deployed by an organization, for example. Our developed management plane offers the service of managing multiple PSC's through a single MSC and our cost analysis reveals that this strategy not just implies more management ease but is more cost effective too II.

A. Data-Filtering Service

A first service that we have designed in the management layer is an authorization mechanism to allow or deny access to data available in smart contract. This data-filtering service is provided through a smart contract based management interface consisting of the MSC's. The PSC is extended to include management operations. These management operations are managed by a MSC in the smart contract based management interface through a **Data Access Manager**. The deployed address of the PSC is provided in the MSC whereas the account address of the Data Access Manager is provided in the PSC. The Data Access Manager is responsible for managing the access of data as per the role / ID assigned to the application / smart contract user through the management operations. The Data Access Manager can update the role/ ID of any user at any given moment of time restricting or permitting data access thereby.

B. Monitoring

The second designed management service is a monitoring mechanism to collect data from a PSC that could be offered through dashboards to an administrator. The monitoring service employs a monitoring interface that comprises of monitoring scripts as depicted in Fig. 5. These scripts monitor the changes in data in the PSC and display the real time values to the users through a dashboard. Changes in private data are sent as email notifications to the administrator of the PSC. The changes in data are displayed at periodic intervals, *e.g.* every 10 seconds. The polling frequency can be adjusted depending on the number of users of the PSC or the application and its popularity. The PSC was extended to include monitoring operations as per need. Latency and code overhead to include the monitoring operations was observed V-B1.

IV. Implementation of the Management Plane

The implementation of the data-filtering service was done in the blockchain itself where the MSC is deployed referencing the deployed address of the PSC. The PSC was extended to include management operations to interact with the MSC. Solidity programming language was employed for programming the MSC and extending the PSC with management operations. The account address of the Data Access Manager was provided in the MSC to restrict updating of access control by non-authorized blockchain users. An option to change the Data Access Manager was also given. The programming code in solidity for a basic MSC is shown in Listing 1 and can be used as a template to develop further more customized MSC. Listing 1 shows a *secondary* MSC that manages the *primary* PSC extended with the management operation ***updateID***.

Listing 1: Solidity based template of a Managing Smart Contract (MSC).

```
pragma solidity ^0.4.6;

contract Secondary{

 function updateAccess(uint ID, uint
     roleno) {
     address Primary=0
        x692a70d2e424a56d2
        c6c27aa97d1a86395877b3a;
     Primary.call(bytes4(keccak256("
        updateID(uint256,uint256)"))
        ,ID,roleno);
       }
}
```

The implemented monitoring service relies on monitoring operations available in the PSC and a set of external scripts that interact with it to collect the data. $Geth$ is the command line interface for running a full Ethereum node. Ethereum provides a JavaScript Console to interact with the blockchain.

This JavaScript console is a REPL (Read, Evaluate and Print Loop) exposing the JSRE (JavaScript Runtime Environment). A monitoring tool was programmed to interact through the JavaScript console to the target deployed PSC. The monitoring tool accesses the JavaScript console through a non default IPC endpoint. The *geth attach* command is used in our prototype to achieve the above. The monitoring tool is implemented through a series of shell scripts. The tool sends message call transactions to the Ethereum network, which are directly executed in the EVM but without effect on the network. Since the call transaction does not broadcast or publish anything on the blockchain, there is no consumption of ether. It is a read-only function that causes no state changes. The return values of these calls are displayed to the application users or the users of the smart contract through a dashboard. Only the public data is displayed through the dashboard whereas the private data, on any change, is notified by an email to the administrator of the smart contract.

Listing 2: A monitoring Script to interact with a Primary Smart Contract (PSC).

```
#!/bin/bash
geth attach ipc:/home/testnet/.
    ethereum/rinkeby/geth.ipc << EOF
    | grep "RESULT:" | sed "s/RESULT:
    ␣//"
primary = web3.eth.accounts[0];
var MyContract = web3.eth.contract(
    ABI);
var MyContractInstance = MyContract.
    at("0x7605a157861e56be8b67c09d␣
    fd5f93f3f3ac0995");
txhash=MyContractInstance.MinDeposit
    .call({from: web3.eth.accounts
    [0]});
console.log("RESULT:␣" +txhash);
EOF
```

The code of a shell script making calls to the blockchain to monitor data is shown in Listing 2. This shell script first employs **geth attach** to connect to the Ethereum public testnet, Rinkeby to allow for further interactions between the script and the blockchain. The account address to be used for sending message calls is first defined and then the address of the deployed smart contract is given. Application Binary Interface (ABI) of the deployed smart contract is needed which is passed on to the script. We only indicate **ABI** instead of listing the ABI of the smart contract for easier reading of the script. Thereafter a message call **MinDeposit** is sent to Rinkeby and the result displayed in console.

V. Experimental Evaluation

A prototype of our management plane for Ethereum based smart contracts is developed using Solidity programming language [7], JavaScript API, [8], bash, SMTP [9] and HTML language. The smart contracts under study were deployed and tested on the Ethereum public testnet, Rinkeby [10].

Table I: Cost analysis of data-filtering in 5 categories smart contracts.

Category	Smart Contract	Ether	Gas Used
Finance	crowdSale.sol	0.01596964	798482
	with data-filtering	0.02447288	1223644
Finance	SquareEx.sol	0.001607982	1339985
	with data-filtering	0.0113376036	1866480
Game	simpleDice.sol	0.01818752	909376
	with data-filtering	0.0325259	1626295
Library	DateTime.sol	0.01885978	942989
	No data-filtering needed	NA	NA
Library	strings.sol	0.00137306	68653
	No data-filtering needed	NA	NA
Notary	notareth.sol	0.01383504	691752
	with data-filtering	0.02718754	1359377
Wallet	SimpleWallet.sol	0.01192706	596353
	with data-filtering	0.02020668	1010334

A. Data-Filtering

We validated and evaluated the management operations on 5 categories of smart contracts, which are **finance**, **game**, **notary**, **library and utilities** and **wallet**. A smart contract was chosen from each category and modified to facilitate data-filtering by extending the contract code with management operations. During this phase, we observed that different categories had different management requirements that we will discuss in the next subsection.

1) Cost Analysis: In order to evaluate the usability versus the overhead associated with implementing data-filtering it was essential to analyze the cost of deployment of the PSC and then the PSC extended with management operations together with the MSC. The MSC is the one that is owned by the *Data Access Manager*. The table I lists down the various costs as per the category of the smart contract. In **finance** category two smart contracts were analyzed as the first needed very little management operations to be implemented. Similarly in the category **library and utilities** two smart contracts were analyzed and none needed any management operations and data-filtering as per our analysis. Table I depicts the cost in Ethers and the gas used for one analyzed PSC in each category. Post data-filtering (indicated as **with data-filtering** in Table I) includes the costs in Ethers and gas used for the PSC with implemented management operations and the MSC.

A single MSC was developed to manage all the considered PSC's to analyze the feasibility and costs of a one-to-many management approach. A comparison was made between the costs of deploying this single MSC managing 5 extended PSC's with the cost of deploying 5 individual MSC's for each extended PSC. The extended PSC is the smart contract that has been extended to include management operations. The comparison results are depicted in Table II. We observe that both the gas used and cost in ether of using a single MSC to

Table II: Cost analysis of management of 5 smart contracts.

Management Relationship	Ether	Gas Used
One-to-Many	0.026072	1303600
One-to-One	0.03246534	1623267

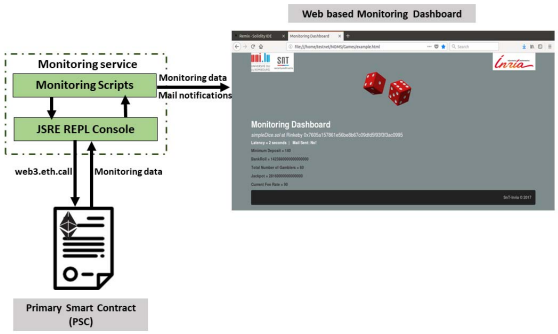

Figure 5: The components of the monitoring service.

manage 5 extended PSC's was less than the overall cost when using single MSC for managing each of the same 5 extended PSC's.

B. Monitoring

The monitoring tool was implemented to display the public data of the *simpleDice.sol* PSC deployed on the public Ethereum, Rinkeby testnet. Experimental evaluation of the tool was done by executing the monitoring script 10 times to monitor PSC data at periodic intervals of 10 seconds. A script was coded to inject data into the PSC at periodic intervals of 5 seconds and this was executed 8 times. Both the scripts were run in parallel. The time difference and also the difference in the number of executions was done to take the mining operation into account, which is not instantaneous in Ethereum. As mentioned before in the previous section, elaborating the design of the management plane *geth attach* was used through the ipc endpoint to interact with Rinkeby through the JavaScript console. Bash script was used to fetch the results of executing *web3.eth.call* [8]. The monitoring script controlled the access of the return values and sent the values to another script, which displayed the return values through a dashboard to the users. A screenshot of the dashboard is shown in Fig.5. There are few function calls in the PSC, where the return values are private and any change in them is intimated to the administrator by using SMTP-based *gmail* notifications, whereas the dashboard simply indicates whether a mail was sent to the administrator during this particular cycle of fetching data or not. A demo of the dashboard is available at [11]. The monitoring tool was implemented for just one PSC but can be extended to any number of PSC's. The basic template that was developed can be applied to any type and any number of smart contracts, on any of the ***Ethereum testnets*** and the main Ethereum network ***Homestead***.

Table III: Code overhead for *simpleDice.sol* smart contract.

Script Function	Lines of Code
Sends Message Calls	16
Monitors Data	37
Displays Data	54

1) Code overhead: The monitoring tool also records the latency observed in fetching all the public data from the PSC including sending the email notification in case of any changes. The latency would vary with the number of message calls to the blockchain network. In our case the monitoring tool sends 6 message call transactions to Rinkeby, which includes monitoring a single private data for email notification in one cycle. It was observed that the latency for one cycle varies between 5 to 7 seconds. The code overhead in terms of the number of lines of code needed to implement the above is given in Table III. Table III examines the script used indicating its function. The total overhead from table III for monitoring the PSC is **107** lines of code as opposed **16** lines of code for just fetching data through 6 message calls. The code overhead is to give a general indication and there can be some reduction in the overhead if the code is optimized.

VI. Discussions

Distributed database as in a blockchain pose significant challenges of security, data access, monitoring and configuration issues. The very same issues that are present in centralized DBMS become more complex in distributed databases. The paper successfully designs, implements and analyzes two crucial components of a management plane namely, data-filtering and monitoring for an application on a distributed network. This is a very valuable initiation towards developing a full-fledged management plane for blockchain and other distributed databases.

Smart contracts are immutable once deployed on the blockchain and no changes can be made in the source code. We have analyzed smart contracts from five different categories. As a result of the analysis it can be concluded that financial smart contracts needed little management operations as the original developers had taken care to include some role-based access to data. Library and utilities needed no data-filtering. In all the smart contracts observed, apart from one, none had the *self-destruct(address)* option. The smart contracts cannot be deemed as complete for mass usage as both role-based data access to allow management operations, monitoring operations as well as control operations to bring about aborting of the smart contract were missing. This makes most of the deployed smart contracts redundant for any practical real world usage except for experimentation. One of the probable reason behind this might be that the smart contracts currently in the blockchain network are limited in the number of users as compared to traditional web applications leading to less focus on implementation by the relatively small community of smart contract developers.

The cost analysis of the management operations that we implemented to provide data-filtering service revealed that smart contracts from the category **Library** needed no management operations as per our analysis. **Wallet**, **Notary**, **Finance** and **Game** needed approximately the same number of functions to implement management operations with the **Finance** smart contracts leading in the amount of gas used followed by **Game**. It was also found through experimental evaluation that a management relationship where a single MSC manages and monitors multiple PSC's is better from an economic point of view in terms of gas usage.

Monitoring of data involves a code overhead and latency that increases with the increase in the number of message calls to the blockchain network, which in turn depends upon the number of functions that return data, being monitored.

VII. Related Work

In [12] Clack *et al.* focus on management of the complete lifecycle of 'smart' legal contracts, whereby they study the creation of legal contract templates and their subsequent use between contracting parties. They do not go into analyzing the management of the smart contracts itself, which has been accomplished in this work. In [13], Gervais *et al.* introduce a novel blockchain simulator to analyze the security and performance of proof of work blockchains like Ethereum. This work differs in being an analysis of a management plane for Ethereum applications employing the use of smart contracts through programmed scripts and developed tools interacting with the real Ethereum public testnet. The work [14] provides a GUI-based tool for users to easily create more secure smart contracts. They also provide a set of design patterns as plugins for developers to enhance security and functionality of the smart contract. Our work is complementary where we provide data-filtering and monitoring templates for developers which can be used for implementing a management plane in applications using smart contracts. In [15] Anderson *et al.* categorize the Ethereum transactions into currency transfers and contract creations whereas this work deals with analyzing a smart contract in each category from amongst the total categories in which smart contract can be demarcated according to the work [5] by Bartoletti *et al.*. In the latter work, the authors make a quantitative analysis of the usage and programming of a smart contract in Ethereum. In [16] Cook *et al.* develop DappGuard to monitor incoming transactions for any smart contract it manages with the intent to detect any potential vulnerabilities.

VIII. Conclusions and future work

In this paper we designed, implemented and evaluated a management plane for both smart contracts and smart contract based decentralized applications. Two services have been instantiated in this plane which are data filtering for access control and data monitoring of deployed smart contracts. The two services could be applied using a one-to-one or one-to-many management strategies for interacting with the managed contracts. Our evaluation of these strategies regarding Ether fees and the used gas shows that one-to-many has a lower cost than a one-to-one strategy.

As a future work, we are working on the extension of the monitoring service with smart contracts based interfaces to monitor deployed ones. In the present implementation the restriction in data access through a decentralized application can be bypassed on the Ethereum blockchain and we aim to circumvent this in our future work. We are also interested in providing a more seamless instrumentation functions for smart contracts to integrate monitoring operations with a little effort from the developers. Finally, we want to investigate further the evaluation of the cost of management while varying different factors including monitoring polling frequencies, number of monitored and accessed attributes, etc.

References

[1] M. Lansiti and K. R. Lakhani, "The Truth About Blockchain, harvard business review," https://hbr.org/2017/01/the-truth-about-blockchain, 2017, accessed: 2018-01-04.

[2] J. O. Julianne Harm and J. Stubbendick, "Bitcoin vs Ethereum," http://www.economist.com/sites/default/files/creighton_university_kraken_case_study.pdf, accessed: 2018-01-04.

[3] "CNBC," https://www.cnbc.com/2017/12/29/ripple-soars-becomes-second-biggest-cryptocurrency-by-market-cap.html, December 2017, accessed: 2018-01-04.

[4] "White Paper," https://github.com/ethereum/wiki/wiki/White-Paper, accessed: 2018-01-04.

[5] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," *CoRR*, vol. abs/1703.06322, 2017. [Online]. Available: http://arxiv.org/abs/1703.06322

[6] B. Radle, "What is RASM ?" http://www.ni.com/white-paper/14410/en/, accessed: 2018-01-10.

[7] "Solidity Documentation," https://solidity.readthedocs.io/en/develop/, accessed: 2018-01-04.

[8] "web3 JavaScript app API," https://github.com/ethereum/wiki/wiki/JavaScript-API, accessed: 2018-01-04.

[9] "SendEmail man pages," http://manpages.ubuntu.com/manpages/wily/man1/sendEmail.1.html, accessed: 2018-01-04.

[10] "Rinkeby," https://rinkeby.etherscan.io/, accessed: 2018-01-04.

[11] "Demo of the Monitoring Dashboard," https://youtu.be/kA7A9ysvT28, accessed: 2018-01-04.

[12] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *CoRR*, vol. abs/1608.00771, 2016. [Online]. Available: http://arxiv.org/abs/1608.00771

[13] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 3–16. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978341

[14] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," *CoRR*, vol. abs/1711.09327, 2017. [Online]. Available: http://arxiv.org/abs/1711.09327

[15] L. Anderson, R. Holz, A. Ponomarev, P. Rimba, and I. Weber, "New kids on the block: an analysis of modern blockchains," *CoRR*, vol. abs/1606.06530, 2016. [Online]. Available: http://arxiv.org/abs/1606.06530

[16] A. L. Thomas Cook and J. H. Lee, "DappGuard: Active Monitoring and Defense for Solidity Smart Contracts," MIT, Tech. Rep., 2017.