Survey of Formal Verification Methods for Smart Contracts on Blockchain

Yvonne Murray[1] and David A. Anisi[1,2]

***Abstract*—Due to the immutable nature of distributed ledger technology such as blockchain, it is of utter importance that a smart contract works as intended before employment outside test network. This is since any bugs or errors will become permanent once published to the live network, and could lead to substantial economic losses; as manifested in the infamous DAO smart contract exploit hack in 2016. In order to avoid this, formal verification methods can be used to ensure that the contract behaves according to given specifications. This paper presents a survey of the state of the art of formal verification of smart contracts. Being a relatively new research area, a standard or best practice for formal verification of smart contracts has not yet been established. Thus, several different methods and approaches have been used to perform the formal verification. The survey presented in this paper shows that some variant of model checking or theorem proving methodology seems to be most successful. However, as of today, formal verification is only successful on simple contracts, and does not support more advanced smart contract syntax.**

***Index Terms*—formal verification, formal methods, theorem proving, model checking, smart contracts, Ethereum, blockchain**

I. Introduction

As it is nearly impossible to alter what has been added to a blockchain, it is crucial that smart contracts are written correctly from the beginning. Formal verification is a tool to verify the logic of a code and make sure that it works in accordance with specification in every situation. To contrast, traditional test-based quality assurance has the disadvantage that it is impossible to test for every single input, series of input and outside conditions in open systems. Thus, an unforeseen input, condition or incident could give errors that will never be discovered by normal testing. Formal verification instead checks the logic of a mathematical model, using proofs, theorems and lemmas, in order to prove the validity of a certain mathematical specification. After checking that the logic in the mathematical model is correct, a conforming program can be written. It is a time consuming process, but should be considered for important pieces of code, as for example smart contracts, where the consequence of an error could be great, as manifested in the infamous Decentralized Autonomous Organization (DAO) smart contract exploit hack in 2016 [1].

A. Objective and Contributions

This paper provides an overview of the current state of the art regarding formal verification of smart contracts. After giving some background about smart contracts and formal verification methods in Section II, the research that has been done will be presented and evaluated in Section III. Section IV will sum up the main results and discuss the importance of formal verification for smart contracts, and how it can achieve wider acceptance and adoption. Finally, Section V will give a short conclusion and suggestions for future work.

[1] Dept. of Mechatronics, Faculty of Engineering and Science, University of Agder (UiA), Grimstad, Norway, yvonne.murray@uia.no

[2]Dept. of Technology & Innovation, Industrial Automation Division, ABB, Oslo, Norway, david.anisi@no.abb.com

II. Background

This section will give an introduction to some material that is important for understanding the remaining of this paper. It is expected that the reader has some prior knowledge about basic blockchain functionality and features [2].

A. Smart Contracts

At its foundation, a smart contract is simply a piece of code [1] running in a host environment, setting up an agreement between two or more parties, including conditions that have to be met before execution. When the predefined conditions are met, the smart contract executes to produce the output. A simple example could be using a smart contract for a payment at a specified date. When the date arrives, the predefined condition is met, and the payment is transferred automatically.

By running a smart contract on a distributed ledger like blockchain, it gains all the advantages of such technology and becomes irreversible, safe from tampering and the entire network can inspect the execution. There is no need for a trusted third party. The identities of the parties are kept pseudonymous, but the bytecode of the contract is visible for the entire network.

Smart contracts can be used to automate several different processes, not only payments. Examples of fields that could benefit from smart contracts are insurance, real estate, supply chains, data recording, identity management and voting. With the growth of the Internet of Things, smart contracts could also play a central role in machine-to-machine interaction [1].

Currently, the main platform for smart contracts is Ethereum. There, one of the languages used to script smart contracts is called Solidity, which is a high-level language specifically designed for smart contracts [1]. Solidity is influenced by common languages like C++, Python and JavaScript, and supports features like inheritance, libraries and user-defined types.

The basic steps for execution of a smart contract are:

1) Coding the predefined conditions and the outcomes
2) Adding the smart contract to the blockchain

978-1-7281-1542-9/19/$31.00 ©2019 IEEE

3) Once the conditions are met, the contract executes and triggers the outcomes

During step 1, a formal verification would be beneficial, so that errors do not become permanent on the blockchain.

B. Formal Verification Methods

Formal verification differs from normal testing in the following ways:

- It is performed during the design process, while testing is performed after the code has been written.
- It ensures that the design itself is correct, while testing checks that the intended design has in fact been implemented without errors.
- It is practically impossible to test for every single input, series of input, unexpected events or outside conditions. By formally verifying the system in the design process, it can be verified that the system works as expected in every situation.

There are several different methods for formal verification, each with their own strengths and weaknesses. The most used in the field of smart contracts are theorem proving and model checking.

1) Theorem Proving: Theorem proving is applicable to many different kinds of systems, and is therefore widely used for formal verification [3]. The system is modeled mathematically, and the desired properties to be proven are specified. Then, the verification is performed on a computer, using a theorem prover software. For the verification process, the theorem prover uses well-known axioms and simple inference rules, and every new theorem or lemma needed for the proof are derived from them [3]. Theorem proving is a very flexible verification method, since it is applicable on all systems that can be expressed mathematically [3].

Theorem provers can be interactive, automated or a hybrid between the two [3]. As the names suggest, the automated ones perform the theorem proving automatically, while the interactive ones might require some human input. For both of them, the system model and the specifications must be set up manually. The hybrid theorem provers provide the possibility to partition the system model based on the complexity level. Then, the least complex parts can be verified with automated theorem proving, while the most complex parts are verified with interactive theorem proving.

Whether it is possible to use automated theorem proving, or if it has to be interactive, depends on the complexity of the system. The complexity of the system decides how complicated the logic has to be in order to create an accurate model. For some simple systems, propositional logic, or statement logic, can be sufficient. Propositional logic consists of declarative sentences or statements, which can be true or false [4]. Different statements can be combined or modified by using Boolean operators, like *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$), or by using implication or equivalence. Propositional logic is decidable, which means that it can be verified automatically by automated theorem proving. However, it has a limited field of application since it cannot correctly represent all kinds of system. First-order logic contains every element from propositional logic, but also adds the possibility to use quantifiers [4]. The quantifiers used in first-order logic are *for all* ($\forall$) and *there exists* ($\exists$). Additionally, there is the possibility to use predicates, which are functions that return true or false, and to declare constants, function names and free variables. This makes first-order logic more expressive, and suitable for more kinds of system. The use of quantifiers makes it possible to express statements using *some*, *all*, *at least* or *at most*. Figure 1 shows some examples of what can be expressed with first-order logic [4].

All men are mortal	$\forall x\ (x\ is\ a\ man \rightarrow x\ is\ mortal)$
No men are mortal	$\forall x\ (x\ is\ a\ man \rightarrow \neg(x\ is\ mortal))$
Some men are mortal	$\exists x\ (x\ is\ a\ man \wedge x\ is\ mortal)$
There is at least one man	$\exists x\ (x\ is\ a\ man)$
There is at most one man	$\forall x \forall y((x\ is\ a\ man \wedge y\ is\ a\ man) \rightarrow x = y)$
There is exactly one man	$\exists x\ ((x\ is\ a\ man \wedge y\ is\ a\ man) \rightarrow x = y)$
There are at least two men	$\exists x \exists y\ (x\ is\ a\ man \wedge y\ is\ a\ man \wedge \neg(x = y))$

Fig. 1. Examples of statements in first-order logic, from [4].

The expressiveness of first-order logic is also the reason why it is not completely decidable. That means that not all statements expressed in first-order logic can be automatically verified by a theorem prover. As a result, some formulas might have to be verified interactively by the user [3]. To interact with the theorem prover, the user provides proof tactics in Meta Language (ML), which is a functional programming language that represents the higher-order logic. For theorem proving, the most expressive logic is higher-order logic (HOL). Higher-order logic allows the use of quantifiers over sets and functions [4]. It can describe any system, given that its behavior is closed-form, meaning that it can be evaluated in a finite number of operations. HOL is not decidable, which implies that user input and verification is needed in addition to the theorem prover [3].

2) Model Checking: Model checking is a method for formal verification that is suitable for systems that can be expressed by a finite-state model [3]. The verification is performed on a computer, using a model checking software. The user provides the model checker with a finite-state model of the system, and a formally specified property it should possess. The model checker then checks if each state of the model satisfies the specification. Every system scenario is examined. If it does not satisfy the property, a counter example is provided, which shows a run of the system that violates the property. This is useful to identify mistakes and correct bugs. On the other hand, if each state of the model does satisfy the specification, the model is formally verified for that specific property. An illustration of the procedure can be seen in Figure 2.

The model checker itself is automatic, but is dependent on an accurate model and property specification in order to give a usable result.

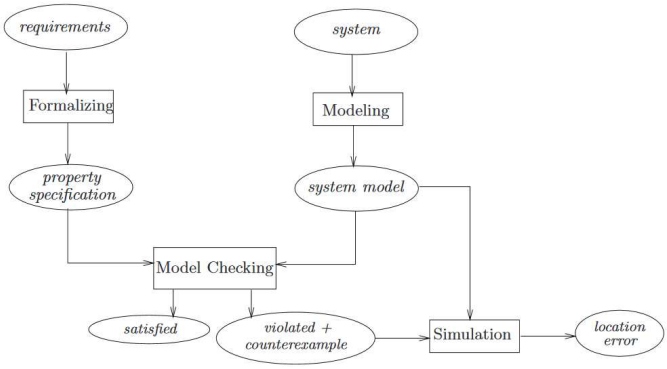

Fig. 2. Procedure for model checking. Figure from [5].

III. State of the Art

This section will present the state of the art for formal verification of smart contracts. The research field is relatively new, which means that there is no established standard for the procedure. Thus, several different approaches have been explored. Since Ethereum is currently the leading platform for smart contracts, most of the research has been focused on formal verification of Ethereum smart contracts, which are mostly written in the language Solidity.

This section is divided into subsections, each presenting an influential and important research paper concerning formal verification of smart contracts. It should be noted that the title of the subsections do not necessarily equal that of the papers.

A. Formal Verification by Translation to F* [6]

A group of researchers from Microsoft Research and Harvard University made an effort to formally verify smart contracts by translating the contract code to F*. They made a prototype of two tools, Solidity* and EVM*, that can assist in language-based verification by translating Solidity and EVM bytecode contracts respectively to F*.

Even though Solidity* translates Solidity to F*, it is important to note that it does not support all Solidity syntax. A specification of the syntax that is supported can be found in [6], and is only a fragment of the complete syntax. Most notably, the translation does not support loops, but does however support recursion. After the translation, F* typechecking is used to detect vulnerable patterns in the code, and it is possible to verify different properties of the translated functions. Out of 396 contracts, 46 were successfully translated and typechecked, and there was discovered problems with reentrancy and some unchecked returns of `send` calls.

EVM* is a tool that decompiles EVM bytecode to F*. The bytecode is the code that is stored on the blockchain, and is therefore often more accessible than the Solidity source code. The translation from EVM* allowed for analysis of low level properties.

The researchers conclude that F* seems to be flexible enough to verify important properties of smart contracts. However, the results are still preliminary, and the suggested approach can not handle all of the Solidity syntax. This is a huge limitation, but as stated in the conclusion of the paper: ”Our approach, based on shallow embeddings and typechecking within an existing verification framework, is convenient for exploring the formal verification of contracts coded in Solidity and EVM bytecode” [6].

B. Towards Formal Verification in Isabelle/HOL [7]

A group of researchers from Australia and France chose an approach for formal verification where the bytecode of the contract was verified in the generic interactive theorem prover Isabelle/HOL. The goal was to create a sound logic for both the requirements and the code. To achieve that, the contracts were split into so-called basic blocks, and the resulting program logic was used for verification.

To split the contract into basic blocks, the bytecode was decompiled in order to extract Control Flow Graphs (CFG). In this case, basic blocks, which consist of code sequences that are not interrupted by jumps, are the vertices of the graph. They are connected by edges, which represent the jumps. In this way, each basic block is a sequential piece of code, where the first instruction is executed first, and it continues uninterrupted until the last instruction of the basic block is finished. The basic blocks are further divided into different types, depending on the last instruction of the block. The types are *Terminal*, *Jump*, *Jumpi* and *Next*. An example of a program split into basic blocks can be seen in Figure 3.

block index	address	instruction	block type
0	0	OR	*Next*
	1	ADD	
	2	SWAP1	
3	3	JUMPDEST	*Jump*
	4	MLOAD	
	5	POP	
	6	JUMP	
7	7	DUP3	*Jumpi*
	8	PUSH1 0	
	10	ISZERO	
	11	JUMPI	
12	12	POP	*Terminal*
	13	RETURN	

Fig. 3. A simplified program split into four basic blocks. Courtesy of [7].

After obtaining the basic blocks, Hoare logic is used as a foundation to create the program logic. Hoare triples are specifically designed for reasoning about program correctness, and is written on the form:

$$\{P\}\ S\ \{Q\} \tag{1}$$

where P is the precondition, Q is the post-condition and S is a program, or in this case a basic block. If we start in a state where P is true and then execute S, S will terminate in a state where Q is true. Using the basic blocks and the Hoare triples, logical rules for the code were formulated at program level, block level and instruction level. As a result, a framework was

created for expressing the EVM bytecode using logic, which is necessary in order to use a theorem prover for verification. The entire verification procedure, from creating a logical model from the code to formally verifying it with Isabelle/HOL, was successfully applied to a case study. However, the framework does not support all of the Solidity syntax. For instance, loops and message calls to other contracts are currently not supported.

C. Model checking of Smart Contracts [8]

In [8], a method for formal verification of a smart contract based application, using model checking is proposed. A symbolic model checker called NuSMV was used. The system model consists of three different layers: 1) Ethereum blockchain, 2) smart contract, and 3) application execution environment.

The Ethereum blockchain model has been greatly simplified as it simply consists of clients, or accounts, and transactions. According to the researches, this simplified model is sufficient for verification of smart contracts. However, to check the behavior over time, the model would also need to add for instance the ledger, blocks, miners and gas. In order to model the smart contract, the Solidity code was translated to the NuSMV input language. A set of translation rules are presented in the research paper to translate simple smart contracts. However, the provided translation is not suitable for advanced smart contracts, and the NuSMV language has some limitations that prevent more advanced rules to be systematized. It is not specified which attributes that separates a simple contract from a complex one. Lastly, the specification to be verified is formalized into logic, and the model checking is performed by NuSMV.

A case study was successfully carried out to illustrate the approach, and five code specifications were formalized in logic, in order to test some key properties. By using the model checking approach, four out of the five properties were formally verified. The last property was not satisfied, and thus a counterexample was presented by the model checker. However, NuSMV has some language limitations that makes it unsuitable for complex contracts.

D. Formal Verification Based on Users and Blockchain Behavior Models [9]

In [9], the authors present a new way of modeling smart contracts and also a way to model the blockchain and user behavior. These new approaches were applied to a name register smart contract, which was then formally verified by using model checking.

The full model used in the formal verification can be seen in Figure 4. In addition to normal user behavior and the contract itself, it models a simple blockchain and hacker behavior. The goal of the hacker is to steal the identity of the user by registering with their own account. Three scenarios are evaluated in order to determine the probability of the user and hacker to successfully register the username. In scenario 1, the user has already registered, and the hacker finds the username from the mined block. In scenario 2, the hacker finds the username from a pending transaction which has not been mined yet. In scenario 3, the hacker finds the username directly from the user call to the contract.

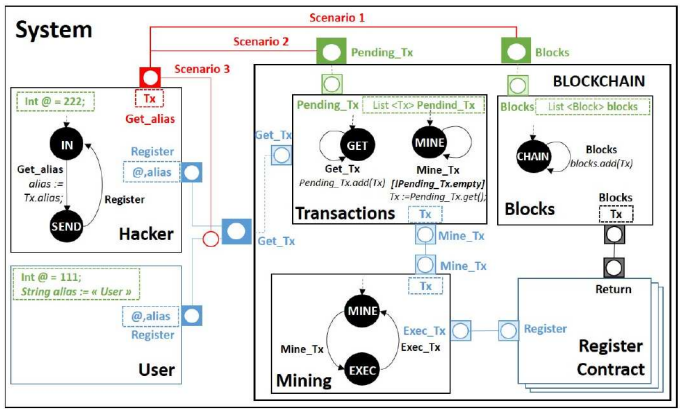

Fig. 4. Model of a user execution of the register smart contract, also taking blockchain and hacker behavior into account. Courtesy of [9].

Using the model checker in Behaviour Interaction Priorities (BIP) to verify the probabilities, the following results were obtained:

- As expected, in scenario 1 the hacker has 0% success probability. The username is already registered for the user, and is immutable on the blockchain. The smart contract behaves correctly and rejects the hacker to register the same username.
- In scenario 2, the hacker has a 12% success probability since the miners decide which transactions to mine, which means that the hacker transaction can possibly be mined first.
- In scenario 3, the hacker has a 25% success probability. The hacker finds the username before the contract has received the call, thus increasing the chances of success.

The conclusion of the researchers is that the modeling technique can provide a good formalization of smart contracts, and that it can be used to find vulnerabilities that would not have been discovered without modeling user behavior. Using a model checker for formal verification, some scenarios vulnerable to hacking were discovered.

E. Verification using Game Theory and Formal Methods [10]

In [10], game theory, formal models and model checking are combined in order to verify smart contracts. It focuses on smart contracts involving physical actions, e.g., internet shopping of physical goods where the buyer pays a deposit, but using blockchain technology and crypto-currency for all the payments. Game theory is used to analyze the behavior of the buyer and the seller, and to see how the choices of one affects the choices of the other. The smart contract itself is modeled formally, and the probabilistic model checker PRISM was used to formally verify certain properties. The actions of the buyer and the seller are uncertain and probabilistic, which is why a probabilistic model checker was used.

The probability to make a certain choice is assumed dependent on the economical benefit. Six different behaviors were

used in the model checking, from honest to gradually more dishonest. Additionally, six different ratios between the deposit and price were used. The model checker was also used to verify specifications given different behaviors. A case where an honest buyer and a quite dishonest seller used the smart contract was evaluated. As expected, the probability for the seller to have a 30% loss was negligible. For the buyer, however, the maximum probability of a 30% loss, at a deposit/price ratio of $\frac{6}{10}$, was 50%.

The conclusion of the researchers was that by adding game theory to the formal verification, the effect of user behavior in smart contracts could be modeled and used in formal verification with a probabilistic model checker.

F. Bug Detection using Formal Methods [11]

Paper [11] regards finding vulnerabilities in smart contracts, using a home made symbolic execution tool called OYENTE. The vulnerabilities discovered were reentrancy, transaction-ordering dependence (TOD), timestamp dependence and mis-handled exceptions. Even though OYENTE was simply used to prove the existence of four common security bugs, not to formally verify the absence of bugs, some elements of formal verification were used in the process. Additionally, with a functional tool for symbolic execution, the first step towards formal verification has already been made. Thus, even though formal verification was not the main goal of the research, it still deserves to be mentioned in this setting.

Using OYENTE, the program variables of the contract are replaced by symbolic variables, and the different paths of the program can be followed and examined. The required inputs to OYENTE are the contract bytecode and the Ethereum current state. Then, by performing symbolic execution, three common contract security problems are discovered, if present. OYENTE also generates the CFG of the contract bytecode, which could further be used for formal verification as described in Section III-B.

All smart contracts in the first 1,459,999 blocks of Ethereum were checked for vulnerabilities using OYENTE. This equals to 19,366 smart contracts, out of which 8,833 were found to have security vulnerabilities.

IV. Discussion

Table I gives a summary of the research done on formal verification of smart contracts. However, it should be noted that the table does not include the work that has been done to formally verify one specific contract, e.g. [12], [13], but rather the verification frameworks that could be applicable to a large number of smart contracts. Several open-source security and bug analysis tools, like [14], have also been disregarded in the survey, as they do not implement any elements of formal methods.

Smart contracts can be an important tool for the future, and can be valuable to many different industries. Because of distributed ledger technology, they can offer immutable and irreversible contracts without the use of a trusted third party. However, before smart contracts can achieve widespread use, they have to gain the trust of the general public. With cases like the DAO attack, the faith in blockchain technology has been shaken. Proving the functionality of the smart contracts using formal methods could be a way to solve this problem of trust.

It is however important to acknowledge that a formally verified contract not necessarily is completely bug free. It is not certain that the specified properties cover all undesirable outcomes. If some important properties are forgotten during the verification, also a formally verified contract can contain mistakes.

Even though formal verification can help increase the trust for smart contracts, it requires a robust framework for the verification itself. Human programmers make mistakes when programming smart contracts. Since humans also play an important role during the verification process, by making models and formulating specifications, also formal verification is prone to errors. Correct models for smart contracts are difficult to obtain, since also the blockchain network and human behavior affects the execution. It is not always that the written code corresponds to the underlying intention or the model corresponds to the actual system. These man-made errors can get undesired consequences. For this reason, it is important to create a robust framework that minimizes the risk of mistakes. It would be useful to develop a library of reusable building blocks and patterns for programming smart contracts that have already been formally verified. This has been done on the blockchain platform Ardor/IGNIS [15], where use of smart contracts is limited to the library provided by the developers. This limits the possibilities for the user, but also limits the possibility for mistakes. Additionally, finding a suitable programming language for both smart contract coding, and later its verification, is of great importance. A language with a strong, unambiguous semantic could make writing error free code easier.

Some work has already been done in order to formally verify smart contracts, as seen in Section III. However, even though formal verification in itself is already a well studied topic, the field of smart contracts is still relatively new, and the research on this application of formal verification is limited. Another consequence of the relatively young age of the research, is that there is no established standard or best practice yet. Thus, several different methods have been explored for achieving formal verification. The two methods that are mainly used in some variant are model checking and theorem proving. The results of the state of the art researches are mixed. Some seem promising for the future, if further developed. However, as of today, formal verification is only usable for simple contracts and simplified models.

V. Concluding Remarks and Future Work

This article has given an overview of the current state of the art for formal verification of smart contracts running on distributed ledgers such as blockchain. At the time of writing, there are merely a few academic papers on the topic, which was

Paper	Verification Method	For all smart contracts?	Model human behavior?	Successful use-case?	Specification tested for	Conclusions
[6]	Translation and type-checking	No, syntax limitation	No	Yes, but not in all instances	Low-level properties, with focus on reentrancy and unchecked calls to `send`.	Promising if it is expanded to work for more of the syntax. At the moment, only 46 out of 396 smart contracts were successfully verified.
[7]	Theorem proving	No, syntax limitation	No	Yes	Low-level properties, like correct returns from different function calls regardless of starting condition	Promising framework for use with a theorem prover, but does not support all Solidity syntax.
[8]	Model checking	No, syntax limitation	No	Yes	Low-level properties, like correct actions triggered by certain circumstances	Promising approach that also models the blockchain environment itself. However, the modelling language can not translate all Solidity syntax.
[9]	Model checking	No, mostly relevant to contracts with human interaction or influence	Yes	Yes	High-level properties, like the ability to sabotage the smart contract for a hacker	Presents a new way to model smart contracts. This method is applied successfully to a specific contract, but it is uncertain if it is also applicable to other kinds of contracts.
[10]	Model checking	No, mostly relevant to contracts with human interaction or influence	Yes	Yes	High-level properties, like probability that a seller or buyer lose their wealth	The addition of game theory enables the modeling of user behaviour in smart contracts, and adds a dimension to the formal verification. However, this approach is only focused on contracts that require human interaction.
[11]	Symbolic execution	Yes	No	Yes	Low-level properties, like reentrancy, transaction-ordering dependence, timestamp dependence and mishandled exceptions	This approach proved the existence of security bugs, but did not formally verify their absence. Out of 19,366 smart contracts, 8,833 had the vulnerabilities that were tested for.

TABLE I
SUMMARY OF THE RESEARCH ON FORMAL VERIFICATION OF SMART CONTRACTS.

expected due to the young age of the technology. However, the development in the field of formal verification of smart contracts is happening fast, and is an active research area. Currently, the approaches can only handle simple smart contracts and simplified models, and are not suitable for complex contracts.

In general, more work is needed on formal verification of smart contracts in the future. The approaches presented in Section III need to be further developed in order to be more widely adopted and usable for complex contracts. One possible example of future work could be to improve Solidity* and EVM* from [6] to support more of the Solidity syntax. In the approaches using model checking, more accurate models of for instance the blockchain behavior would improve the results.

At the moment, formal verification is a complicated process that is not accessible to end-users as an out of the box functionality. Ideally, every published smart contract should be formally verified before adoption, and it would be an advantage if there existed a framework that automated as much of the process as possible. Another possibility could be to create a library of already verified reusable blocks and patterns for smart contracts, similarly to the setup and functionality found in Ardor/IGNIS. It limits the possibilities, but could in some cases be sufficient.

ACKNOWLEDGMENT

The research presented in this paper has received funding from the Norwegian Research Council, SFI Offshore Mechatronics, project number 237896.

REFERENCES

[1] C. Dannen, *Introducing Ethereum and Solidity*. Apress, 2017.
[2] D. Drescher, *Blockchain Basics*. Apress, 2017.
[3] O. Hasan and S. Tahar, "Formal Verification Methods," *Encyclopedia of Information Science and Technology, Third Edition*, 2015.
[4] I. Chiswell and W. Hodges, *Mathematical Logic*. Oxford University Press, 2007.
[5] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.
[6] K. Bhargavan *et al.*, "Formal Verification of Smart Contracts: Short Paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, 2016.
[7] S. Amani *et al.*, "Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018.
[8] Z. Nehai, P. Y. Piriou, and F. Daumas, "Model-Checking of Smart Contracts," in *Proceedings of the IEEE International Conference on Blockchain*, 2018.
[9] T. Abdellatif and K.-L. Brousmiche, "Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models," *IFIP NTMS International Workshop on Blockchains and Smart Contracts (BSC), Feb 2018, Paris, France*, 2018.
[10] G. Bigi *et al.*, *Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods*, pp. 142–161. Springer International Publishing, 2015.
[11] L. Luu *et al.*, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
[12] Y. Hirai, "Formal Verification of Deed Contract in Ethereum Name Service." https://yoichihirai.com/deed.pdf, 2016. A ccessed 2019-03-07.
[13] X. Bai *et al.*, "Formal Modeling and Verification of Smart Contracts," in *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ICSCA 2018, pp. 322–326, ACM, 2018.
[14] "Mythril." Website, https://github.com/ConsenSys/mythril-classic.
[15] Jelurida Team, "Jelurida Ardor/IGNIS Whitepaper," 2017.