# WILL THAT SMART CONTRACT REALLY DO WHAT YOU EXPECT IT TO DO?

TNO innovation for life

**WHITE PAPER**
januari 2018

Maarten Everts
Frank Muller

**Smart contracts** are believed to be the next step in inter-party automation; traditional trusted third parties are to be replaced by small pieces of code running on a distributed system. However, at the end of the day, these smart contracts controlling high-value assets are pieces of code **written by fallible humans**. And because of the special nature of smart contracts, mistakes or bugs can have a significant financial impact, as illustrated by a number of recent **high-profile incidents**. In this white paper we explore this problem further and describe a number of strategies and approaches for the creation of secure and robust smart contracts, some of which can be applied right now, while others require additional research & development to be applicable.

# CONTENTS

# 1. INTRODUCTION

BITCOIN[1] STARTED AS A GRASS-ROOTS PROJECT AND HAS SHOWN HOW ITS UNDERLYING BLOCKCHAIN TECHNOLOGY CAN BE USED TO BUILD A DISTRIBUTED, TRUSTLESS[2] VALUE-EXCHANGE SYSTEM ON THE GLOBAL INTERNET. The Ethereum project[3], seen by some as the second-generation blockchain, built on Bitcoin's ideas to produce a strongly decentralized platform that allows for *programmable* transfer of ownership of value, including virtual assets and representations of real-world assets.

Through so called *smart contracts*, users can precisely codify their agreements and trust relations, which after deployment will automatically be executed by the (Ethereum) platform. The vision for these smart contracts is that they will facilitate economic activity by effectively providing services that are traditionally offered by (trusted) third parties and intermediaries (e.g., banks, notaries, courts).

However, that what makes smart contracts such a powerful concept is also its Achilles' heel. After deployment on the blockchain platform, the smart contract itself cannot be modified or manipulated anymore, which is what gives all participants the confidence to depend on it. But this very same property also makes any mistake in the code of the smart contract very hard (or near impossible) to correct.

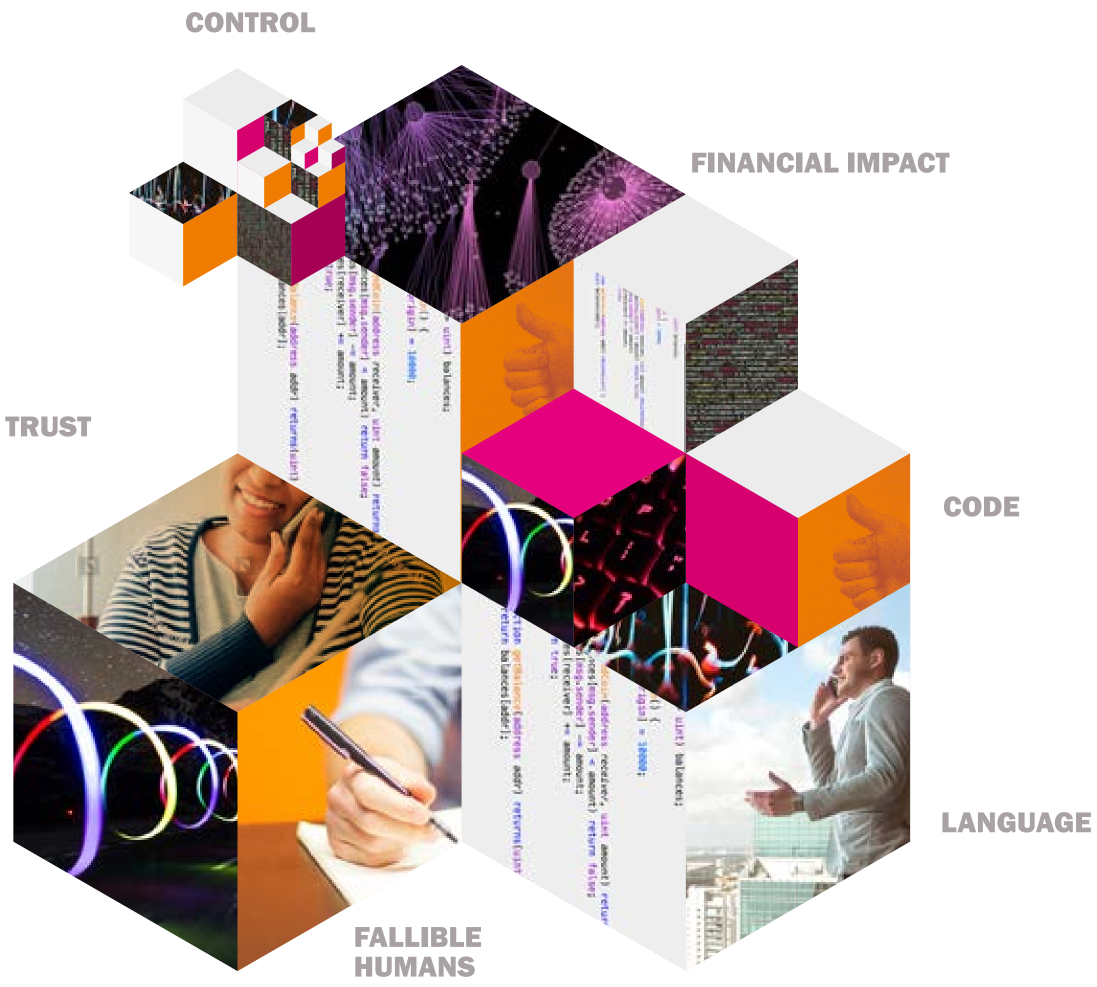
The blockchain world is thus moving towards a future in which we yield the control of high value assets to smart contracts that:
- are unchangeable, autonomous and unstoppable,
- are publicly visible and analyzable,
- run in a public, hostile environment,[4]
- are written by fallible humans.

A number of recent incidents concerning bugs in smart contracts leading to multi-million losses (see Section 3) have shown this is a recipe for disaster and illustrate that authoring smart contracts requires a different set of development practices than the happy-go-lucky style that is common in current mainstream software development [1].

That what makes smart contracts such a powerful concept is also its Achilles' heel.

1 https://bitcoin.org
2 As in, there is no need to trust another participant to do the right thing, as long as the majority is honest.
3 https://ethereum.org
4 For consortium or permissioned blockchains this is not necessarily the case, see section 2.1.

In this paper we will first dive into the world of smart contracts in Section 2 and then continue in Section 3 to explore the ways smart contracts can fail. This is followed by a discussion of a number of strategies and approaches for the creation of robust and secure smart contracts. Do note that while this paper appears to focus on the Ethereum platform, this is mostly because the prevalence and momentum of the platform makes illustrative failure cases and (partial) solutions more abundant than for other platforms.



# 2. BLOCKCHAIN AND SMART CONTRACTS

FROM A TECHNICAL PERSPECTIVE, A BLOCKCHAIN PLATFORM CONSISTS OF A PEER-TO-PEER NETWORK OF A NUMBER OF COMPUTER NODES, OWNED BY MUTUALLY DISTRUSTING DISTINCT ENTITIES, COLLABORATING TO REACH CONSENSUS ON THE ORDER OF CHANGES TO THEIR EVER-CHANGING SHARED REALITY (I.E., A DATABASE). The changes to the shared reality come in the form of transactions, for each of which the participating nodes need to decide and agree whether they conform to the rules agreed upon. In Bitcoin, for example, this shared reality is a ledger of who owns what bitcoin and in Ethereum it is the state of a replicated, though slow, "world computer" that is programmable using so called smart contracts. On top of all this, the user-facing applications are (to be) built. The above concise description of blockchain technology nicely maps to a 4-layer model, as illustrated in Figure 1.

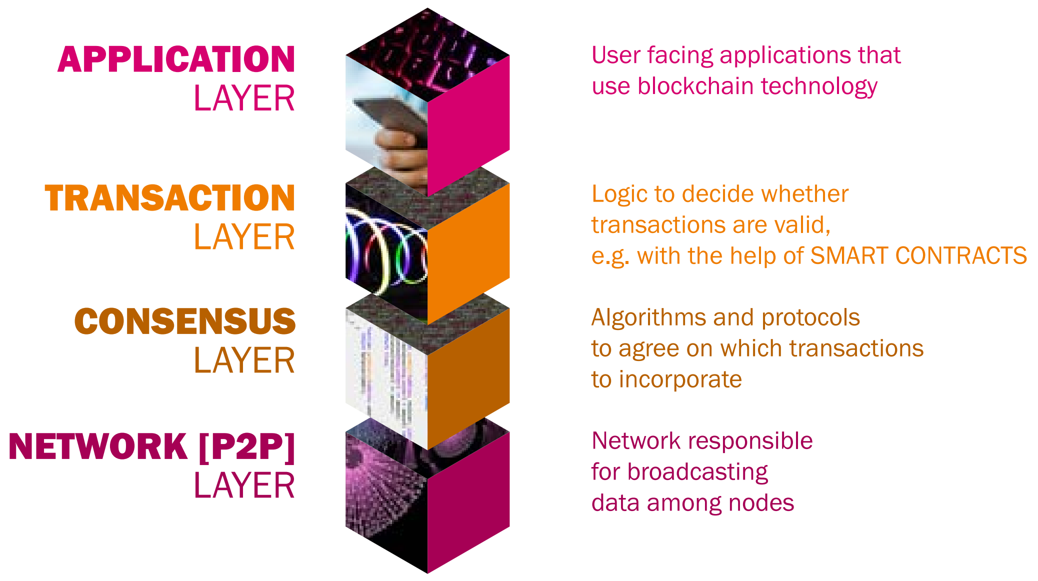

APPLICATION LAYER
User facing applications that use blockchain technology
TRANSACTION LAYER
Logic to decide whether transactions are valid, e.g. with the help of SMART CONTRACTS
CONSENSUS LAYER
Algorithms and protocols to agree on which transactions to incorporate
NETWORK [P2P] LAYER
Network responsible for broadcasting data among nodes

FIGURE 1: A FOUR-LAYER MODEL FOR BLOCKCHAIN TECHNOLOGY

# 3. CONSENSUS IN PUBLIC AND IN CONSORTIA

ON THE PUBLIC BITCOIN AND ETHEREUM PLATFORMS, FITTING WITH THEIR LIBERTARIAN ROOTS, ANYONE CAN JOIN AND PARTICIPATE IN THE CONSENSUS PROCESS, AND NO PERMISSION FROM SOME CENTRAL AUTHORITY IS NEEDED; ONE ONLY NEEDS TO BE WILLING DO SOME (COMPUTATIONAL) WORK IN THE FORM OF FINDING A SOLUTION TO A CRYPTOGRAPHIC PUZZLE.[5] Participants are also compensated for this work with the platform's internal currency, as long as their contribution to the consensus process abides to the consensus rules.

Later, driven by the need for more (regulatory) control, privacy, and capacity, so called permissioned or consortium blockchains started to be developed. In such consortium blockchains, all parties are typically (legally) identifiable and permissions on who can participate are strictly, if not centrally, controlled. Prime examples of such consortium blockchain initiatives are the Hyperledger Fabric project[6] and R3's Corda project[7].

## 3.1 SMART CONTRACTS

MUCH OF THE EXCITEMENT (AND HYPE) SURROUNDING BLOCKCHAIN TECHNOLOGY REVOLVES AROUND THE PROMISE OF SMART CONTRACTS. These (relatively) small pieces of software code are evaluated by all[8] the nodes of a blockchain platform when triggered through incoming transactions. The rules and algorithms in the smart contract can in this way govern the ownership of assets (e.g., digital currency) and other "state" as described in the code. Assuming an honest majority of nodes, the smart contracts will always execute as specified, which is what makes them suitable to take on the role of an efficient, impartial intermediary; all stakeholders can depend on the truthful execution of the code, without any one party having the ability to interfere. Applications range from basic multi-signature wallets and escrow services to full-blown "Distributed Autonomous Organizations" (DAOs), where the latter are virtual companies that are fully governed by code and do not have identifiable owners.

From the start, Bitcoin has had a form of programmability with "Scripts" [2] that allow one to express the rules of how and when funds could be "unlocked", but it is deliberately limited and it was not the main focus of the project. In contrast, for Ethereum programmability was the main selling point when it was introduced [3], [4] as a distributed, blockchain-based platform with a built-in Turing-complete[9] programming language to create smart contracts with.

5 This process called "proof-of-work" is currently used by most public permissionless blockchains, which basically boils down to trying many possible solutions until you find one that fits the requirements. Due to the high (energy) cost of this consensus approach, many are looking for alternatives, of which proof-of-stake is most promiment. In proof of stake, your "voting right" in the consensus mechanism does not depend on your computational power, but the amount of currency you have (and put "at stake").

6 https://www.hyperledger.org/projects/fabric

7 https://www.corda.net/

8 Actually not necessarily all nodes, this depends on the blockchain platform; in particular the consortium blockchains include controls to decide which code is run and seen by which participant for confidentiality and performance reasons.

9 Semi-Turing complete actually; all execution of smart contract code is bounded as the caller of the code has to supply "gas": compensation for the execution payed in the platform currency (ether).

Although both the term and the concept of a smart contract were already introduced in 1996 by Nick Szabo [5], the birth of the Ethereum platform made the term more widespread. Consequently, Ethereum is currently the most prevalent smart contract platform in terms of visibility, momentum, and development tooling. Smart contracts on Ethereum are submitted to the platform in the form of low-level binary bytecode that is to be executed on a special purpose virtual machine (EVM) by all the nodes. The low-level bytecode is typically produced through compilation from a high-level, human readable language, of which Solidity[10] is the most prevalent.[11]

Also in the world of permissioned (consortium) blockchains smart contracts play an important role. Some use Ethereum's EVM and development ecosystem (e.g., Monax[12] and Quorum[13]), while others use traditional general purpose programming languages. For example Hyperledger Fabric uses the Go programming language to program smart contracts in (which they call "ChainCode") and R3's Corda uses Java. At the end of the day though, smart contracts are still code written by fallible humans, which means mistakes are likely to be made.

## 3.2 HOW SMART CONTRACTS CAN FAIL

A SMART CONTRACT, LIKE ANY COMPUTER CODE, ALWAYS SIMPLY EXECUTES AS SPECIFIED, AND FOR THE PLATFORM ITSELF THERE IS NO RIGHT OR WRONG. One talks of mistakes or bugs in code when there is a (strong) mismatch between the underlying *intention, assumption*, or *expectation* and the actual code that is executed. As mentioned in the introduction, such mismatches have in the past already resulted in a number of multi-million dollar incidents.

One talks of mistakes or bugs in code when there is a (strong) mismatch between the underlying intention, assumption, or expectation and the actual code that is executed.

10 https://solidity.readthedocs.io
11 One of the earliest languages for the EVM was "Serpent", a python inspired language that since has been deprecated because of the many (security related) problems in its compiler. [6]
12 https://monax.io/
13 https://github.com/jpmorganchase/quorum

## 3.3 HIGH-PROFILE SMART CONTRACT FAILURES

THE MOST FAMOUS INCIDENT TO DATE WAS IN THE SUMMER OF 2016, WHEN “THE DAO”, A SMART CONTRACT GOVERNED VIRTUAL COMPANY RUNNING ON THE ETHEREUM PLATFORM, IMPLODED AFTER AN EXPLOITABLE BUG HAD BEEN FOUND IN THE SMART CONTRACT CODE. The DAO was an example of a Distributed Autonomous Organization, and in essence, a virtual investment fund, that allowed participants to buy a stake, which would give them proportional voting rights on which future proposals should be funded. To the surprise –and perhaps even dismay of the initiators– The DAO managed to collect nearly $150M worth (at the time) of ether. However, shortly after the crowd-funding period, an attacker managed to gain control over a significant portion of the value in the fund (nearly $60M), due to a bug in the smart contract code [7]. The resolution came in the form of a controversial, coordinated platform intervention that reverted the theft and dismantled the The DAO. Some opposed this intervention, feeling it violated the “code-is-law” principle [8], and pushed for a fork of the platform (see [9] for a back story).

The attacker managed to gain control over nearly $60M in the fund due to a bug in the smart contract code.

Approximately a year later, another smart contract bug resulted in a multi-million dollar theft. The Parity multi-sig wallet allows one to spread control of funds over multiple people, for example by only allowing the transfer of funds when 2 out of 3 pre-defined signatures are presented. Unfortunately, there turned out to be a small mistake in the initialization code, allowing an attacker to trivially gain control of the wallet and divert the funds inside elsewhere. As a number of high-profile crowd-funded projects used this wallet to manage their funds, this simple bug resulted in the theft of approximately $30M [10].

Interestingly, only a few months later there was another multi-million dollar incident with the updated version of the very same Parity multi-sig wallet. This time, $160M worth of ether was accidentally frozen because a bug in a so-called library contract allowed the library contract be triggered to disable and remove itself, locking up the funds of all 500+ wallets that depend on this code [11]. As this issue appears to be less contentious than The DAO incident, it is not unlikely there will be some sort of platform intervention to unlock these funds.

## 3.4 MISTAKES IN SMART CONTRACTS

WHEN WRITING SMART CONTRACTS –AS IS THE CASE FOR ANY OTHER PIECE OF SOFTWARE– THERE IS OF COURSE PLENTY OF ROOM FOR STRAIGHT-UP LOGIC ERRORS, IN PARTICULAR IN MORE COMPLEX SMART CONTRACTS. However, smart contract platforms themselves have a number of specific subtleties and pitfalls that can lead to (potentially exploitable) mistakes (see [12]–[14] for an overview).

On the Ethereum platform for example, a smart contract transferring ether to an account can trigger the execution of code in case that particular account is governed by another smart contract, which can sometimes lead to surprising effects. To illustrate, the bug in the infamous “The DAO” smart contract was a subtle reentrancy issue; neither the authors of the smart contract nor the reviewers had realized that a seemingly benign transfer of ether would allow an attacker to re-invoke the same piece of code in the smart contract, manipulating its internal state more than the intended one time [7]. Similarly, a smart contract author can easily be surprised by the effects of code executed as a result of a transfer of ether if the various forms of exceptions supported by the EVM are not properly handled. Such mishandling of exceptions is what caused the demise of the (not so serious) “King of the Ether Throne” smart contract.[14]

Smart contract platforms themselves have a number of specific subtleties and pitfalls that can lead to potentially exploitable mistakes.

14 See https://www.kingoftheether.com/postmortem.html for a post-mortem.

In addition, one needs to consider the environment the smart contracts “live” in. Even though the invocations of a smart contract are executed by all participating nodes in a blockchain network to make sure it is executed correctly, the node (i.e., miner) that produces the block can sometimes still have some subtle influence on the outcome of a smart contract. In smart contracts with transaction-ordering dependence for example, miners can re-order transactions in a block for their advantage, and in smart contracts with timestamp dependence miners have some flexibility in choosing the timestamp for the block, which for example can be used to influence a timestamp-based lottery [13].

Finally, the problems smart contracts try to solve often include some aspects of “fairness” and game theory, which is something that is still hard to model and put into code. According to some, the aforementioned and now defunct “The DAO” smart contract, also contained a number of game-theoretic flaws besides the fatal flaw that led to its demise [15]. In addition, one needs to remember that by default, all information the smart contract bases its decisions on is public. Keeping certain pieces of information private will typically require the use of authenticated data-sources (oracles) or advanced (zero-knowledge) cryptography [16], which is non-trivial in its own right.

In general, it appears that the Ethereum platform itself, and in particular its Solidity language make writing smart contracts error prone (see also the next section). Atzei et al. [14] provide an excellent systematic survey of attacks on Ethereum smart contracts.

## 3.5 SEMANTICS IN SMART CONTRACTS

BESIDES THE EXECUTION, ANOTHER POTENTIAL SOURCE OF SURPRISE OR CONFUSION IS THE MEANING OR *SEMANTICS* OF THE CONCEPTS THAT A SMART CONTRACT REFERS TO. Rigorous, well-defined semantics are crucial to be able to make a mapping between real-world and digital concepts. The importance of semantics is well illustrated by the Mars Climate Orbiter incident in which the spacecraft was lost on entry to the Martian atmosphere because one of the software components used imperial rather than metric units [17]. While [13] recognized that a “semantic gap” frequently exists and is partially responsible for the failure of smart contracts to execute as expected, their focus is on the semantics of the underlying infrastructure rather than the semantics embedded in the smart contract. Even though there have not yet been (recorded) incidents of smart contract issues caused by semantic confusion, this is bound to happen as smart contracts become more complex and interconnected. There is a wide variety of standards, vocabularies and ontologies available for different domains to formalize the relations and meaning of concepts used (cf. for example schema.org) but so far no research has been undertaken on the integration of such semantic vocabularies into smart contracts.

# 4. TOWARDS ROBUST AND SECURE SMART CONTRACTS

THE ABOVE DISCUSSION ILLUSTRATES THAT BECAUSE OF THEIR PARTICULAR NATURE AND THE ENVIRONMENT THEY “LIVE” IN, SMART CONTRACTS REQUIRE A DEVELOPMENT PROCESS THAT IS DIFFERENT FROM TRADITIONAL SOFTWARE DEVELOPMENT. It is more akin to the development of safety critical software or the development of hardware (silicon). In fact, the guidelines, tooling and wisdom from these fields already provide a solid basis to draw inspiration from for the development of secure and robust smart contracts.

In this section we highlight a number of strategies and approaches for the creation of secure and robust smart contracts that can be applied right now as well as approaches that require additional research and development.

## 4.1 BEST PRACTICES

WRITING SECURE SOFTWARE IS NOT A ‘NEW THING’ AND THERE ARE ALREADY GENERAL SOFTWARE DEVELOPMENT BEST PRACTICES THAT ARE VERY APPLICABLE TO SMART CONTRACT CREATION. This includes among others, risk analysis, collection of security requirements, identification of abuse cases, and a proper definition of the attacker model. However, due to their nature and the (hostile) environment they live in, smart contracts require extra, if not special, attention. In addition, there are a number of platform-specific pitfalls to avoid.

After the high-profile security incidents, the Ethereum community started focusing more on the security of smart contracts. A number of blog posts have been dedicated to the subject [18]–[20] and various security guidelines have been published, both inside the official Solidity documentation [21] and elsewhere [22]. Some guidelines are very platform (Ethereum) specific, while others are more general. We will highlight a number of them in this section and the following.

A first general step is simply keeping smart contracts small and simple. In addition to a reduced attack surface, having a smaller amount of code makes it easier to reason about and scan for potential vulnerabilities. This comes down to consciously deciding what aspects of the business logic should be put in the smart contract and what parts can safely be pushed towards the ‘edges’ of the platform. Only those aspects that relate to security and trust (or the lack thereof) should be incorporated in the smart contract.[15] Of course, as discussed elsewhere [22], there are subtle trade-offs to consider in terms of simplicity, reusability and being able to reason about code.

15 Some parallels can be drawn to so called applets running on smart cards. Like smart contracts, smart cards have limited computing power (low power) and they provide security services. And similarly, mistakes in smart card applets can have a high financial impact and are non-trivial to patch.

Writing secure software is not a 'new thing' and there are already general software development best practices that are very applicable to smart contract creation.

General software development best practices that are also very suitable for smart contracts include defensive programming, fuzzing, and use of automated tests and test frameworks. For example, Truffle[16] is a development and testing framework for Ethereum (Solidity) smart contracts.

Furthermore, as with traditional software, code reviews [23] and code audits[17] can prove very valuable in preventing bad smart contract code from going into production. However, such audits can be very labor intensive and thus expensive. Therefore, we should look for automated tooling to at least prevent the most common pitfalls.

## 4.2 STATIC ANALYSIS TOOLING FOR SMART CONTRACTS

EVEN THOUGH COMPUTER SOFTWARE IS (STILL) NOT CAPABLE OF DEDUCING A USER'S INTENT, THERE ARE PLENTY OF COMMON MISTAKES AND PITFALLS THAT ARE "MACHINE DETECTABLE". Such static analysis tools can, besides finding potential mistakes, also help in simply getting a better understanding of the code's behavior.

16 http://truffleframework.com/
17 There are already firms providing audit services for smart contracts, for example https://zeppelin.solutions/security-audits.

In practice, the compiler itself already forms the first line of defense; a proper type system prevents a whole class of bugs and typically compilers issue warnings for expressions that are most likely to be mistakes (e.g., the use of the assignment operator “=” in an if statement instead of the comparison operator “==”). Static analysis tools take this a step further and look for more and larger “patterns” of potential bugs.

Static analysis tools can be applied to both the human-readable smart contract language and the more low-level bytecode. Application to the first has the advantage that it gives the tool more (contextual) information and that it is able to give the user better feedback how to correct potential errors, while application to the latter is also possible when the original source is not available, which in the case of Ethereum is not uncommon. In addition, static analysis on bytecode has the advantage it analyses the code that is actually being run, circumventing potential bugs in the compiler.

A number of analysis tools for Ethereum’s Solidity language have been developed in the past few years. Solium[18] for example is a so called linter[19] for the Solidity language that can detect predefined potential problematic patterns in the abstract syntax tree of a given smart contract. Solgraph[20] uses a more visual approach by producing control-flow graphs to help detect potential (security) problems.

There are plenty of common mistakes and pitfalls that are “machine detectable”.

Similarly, analysis tools have started to become available that analyze the low-level (EVM) byte code directly. Porosity[21] for example, is an open source decompiler that takes EVM byte code and turns it into Solidity code that is more palatable for human inspection for (security) issues. The tool itself also already checks for potential security issues. Oyente[22] is another open source static analysis tool for EVM byte code and accompanied by an academic paper [13]. It is based on symbolic execution of a subset of the EVM (called EtherLite), which allows it to thoroughly check with some heuristics for a number of predefined security problems, including transaction-ordering dependence, timestamp-dependence, and mishandled exceptions. The open source Mythril project[23] is a recent addition to the static analysis toolbox. Like Oyente, it can check EVM byte code for a number predefined security issues. In addition, it provides basic visualizations of control flow.

18 https://github.com/duaraghav8/Solium
19 A linter is a tool for detecting and flagging errors and suspicious language usage in source code, including stylistic errors.
20 https://github.com/raineorshine/solgraph
21 https://github.com/comaeio/porosity
22 https://github.com/melonproject/oyente
23 https://github.com/b-mueller/mythril/

In addition to these stand-alone tools, companies are also starting to provide online services that can automatically check for potential vulnerabilities in smart contracts. Securify[24] for example is providing an automated online analysis tool to find known critical security vulnerabilities and typical coding mistakes, partially based on formal method technologies (see next section). Quantstamp[25] takes this a step further and aims to provide a (token-based) platform for incentivized manual and automated audits for securing (Ethereum) smart contracts.

As mentioned above, exploitable bugs are the result of a mismatch between what is expected and what is actually defined, which is the gap such analysis tools aim to bridge. However, in order to do this, static analysis tools need some kind of definition of the semantics of the platform. And preferably, both a formal definition of the semantics [24] and a formal specification of the intent, which brings us in the world of formal methods.

24 http://securify.ch/
25 https://quantstamp.com/

## 4.3 FORMAL VERIFICATION OF SMART CONTRACTS

DETERMINING WHETHER OR NOT A PIECE OF (SMART CONTRACT) CODE MATCHES ONE'S EXPECTATIONS IS KNOWN TO BE UNDECIDABLE. But if the code would be accompanied by rigorous mathematical proofs that show that certain high level properties always hold, this will increase the confidence in said code, as long as the high level properties, as defined in the specification, connect well to the expectations. Or to put it in other words, the gap between intention and proven high level properties is likely to be smaller. Even though theory tells us it is not possible to automatically prove the correctness of all possible programs in general, it is possible to prove the correctness of many useful ones.[26] Formal methods is the field that works on such formal proofs for code and because of the relative small size of typical smart contracts, they are the perfect landing ground for recent academic advancements in this area. This is also recognized in the Ethereum community and several initiatives are under way [25].

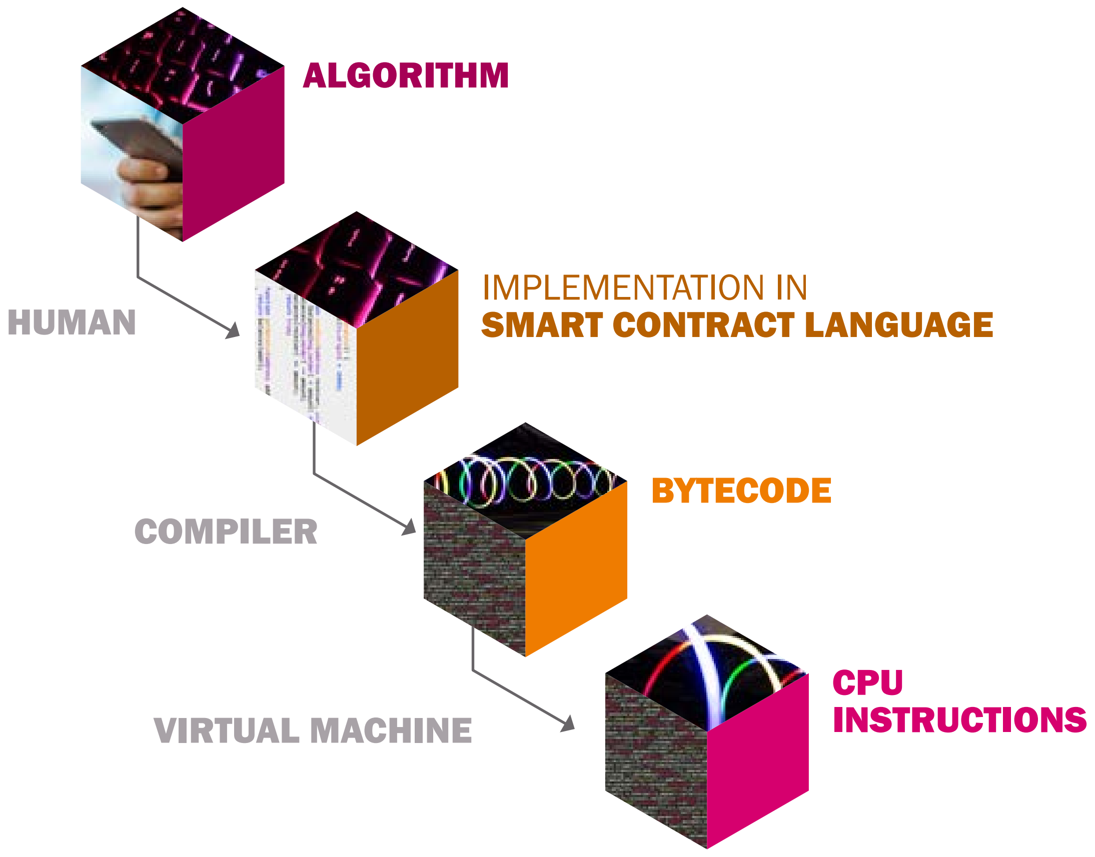


FIGURE 2: CONCEPTUALLY, A SMART CONTRACT EXISTS IN DIFFERENT LAYERS, AND FOR EACH LAYER FORMAL METHODS CAN HELP IN UNDERSTANDING.

26 See also the presentation by Andrew Miller, "Ethereum Isn't Turing Complete, and It Doesn't Matter Anyway" (https://youtu.be/cGFOKTm_8zk).

**If code would be accompanied by rigorous mathematical proofs that show that certain high level properties always hold, this will increase the confidence in said code.**

A smart contract conceptually exists at various 'layers'. At the highest layer it is a non-executable design or algorithm that may or may not be formalized. Often there is an intermediate, human readable form expressed in a smart contract language (in case of Ethereum typically Solidity), which is then translated into byte code by a compiler. The byte code itself is then interpreted or compiled (just-in-time) by a virtual machine (EVM in case of Ethereum) to execute the smart contract on the CPU[27] of each of the nodes in the blockchain network (see also Figure 2). And for each of these layers, formal methods can play a role in either increasing confidence of a specific smart contract or increasing confidence in the platform as a whole.

Formally specifying the algorithm to use in a smart contract and the accompanying invariants (i.e., specification), will already increase the confidence in the design and unearth tricky edge-cases. Furthermore, formal method tools like TLA+[28], created for the specification and verification of concurrent and distributed systems, appear to map well to the multi-transactional behavior of smart contracts, as explored in a recent paper by Sergey et al. [26]. For example, they show how The DAO's reentrancy issue is an instance of "concurrentesque" behavior that can be modeled and analyzed as such. We believe that specifying and analyzing smart contracts in tools like TLA+ is something to be explored further and has the potential to prevent many problems and uncover issues in already deployed smart contracts.

Of course, ideally, one would want the smart contract code or even the low-level byte code to be accompanied by similar proofs of the smart contract's high-level properties. Such certified code can provide a much more "end-to-end" proof of correctness, provided of course both the compiler and the VM (or interpreter) executing the smart contract code are also correct. A number of efforts in the Ethereum community have started along this path.

One of the first efforts was an addition to the standard Solidity compiler [27] that allows Solidity smart contracts to be annotated with Hoare-style pre-/postconditions. The compiler translates this to an intermediate functional language (ML) to be able to apply the Why3 tool [28] for the verification of basic safety properties. However, an approach like this assumes that the compiler is correct and produces the expected EVM code, which in the case of Solidity has already been shown to be an issue.[29]

27 Of course, there are many more layers below this level.
28 http://lamport.azurewebsites.net/tla/tla.html
29 At least two bugs in the Solidity compiler have had potential security implications, see https://blog.ethereum.org/2017/05/03/solidity-optimizer-bug/ and https://blog.ethereum.org/2016/11/01/security-alert-solidity-variables-can-overwritten-storage/.

The next step is to also prove high-level properties of low-level byte code. Bhargavan et al. [29] use the programming language and verification framework F*[30] in a two-pronged approach. They provide both a tool that converts a subset of Solidity to F* and a decompiler that translates EVM code to F*. The latter can not only be used to analyze smart contracts for which the source code is unavailable, it also allows for the equivalence proofs between a Solidity program and the byte code output of the Solidity compiler.

Even though theory tells us it is not possible to automatically prove the correctness of all possible programs in general, it is possible to prove the correctness of many useful ones.

A different approach is taken by Ethereum's formal methods engineer Yoichi Hirai. He formalized the semantics of the Ethereum Virtual Machine (EVM)[31] in Lem[32], which can be compiled to specifications that can be used with theorem provers such as Coq[33] and Isabelle/HOL[34]. It is still work-in-progress, but he used it already to produce a limited safety proof for a relatively small and simple smart contract [30].

Similarly, the KEVM project[35] also started from the semi-formal yellow paper [4] to create an executable and human readable model of reference semantics for EVM programs. It uses the K framework[36], which is a rewrite-based executable semantic framework. The process of producing these reference semantics already uncovered a number of ambiguities [24] in the Ethereum Yellow Paper [4]. In addition, the reference semantics were used to produce an executable EVM interpreter that passes all the existing official EVM stress tests for compliant EVM implementations. Furthermore, the accompanying paper [24], shows how the reference semantics can be used to produce analysis tools automatically.

30 https://www.fstar-lang.org/
31 https://github.com/pirapira/eth-isabelle
32 https://www.cl.cam.ac.uk/%7Epes20/lem/
33 https://coq.inria.fr/
34 https://isabelle.in.tum.de/
35 https://github.com/kframework/evm-semantics
36 http://www.kframework.org

Of course, formal methods are also no silver bullet. First of all, formal methods can only check for those properties that are actually specified. Besides mistakes in specifications, if important and relevant properties and invariants are not specified, one can still be surprised by the “formally proven” smart contract code. For example, consider a smart contract for governing a crowd funding process. If one omits to specify that a participant should never be able to back out with more than originally contributed, then no formal verification will be able to detect mistakes in the code that would allow for the violation of such a relatively obvious constraint. In addition, writing proofs is still hard. While the fact that smart contracts are relatively small helps somewhat in that regard, it is still a non-trivial endeavor for which there is room for additional research and development. Furthermore, this calls for the development of libraries of re-usable patterns and building blocks for smart contracts with formally proven properties, both at the algorithmic level and at the implementation level (see also the section below).

Preferably, formal verification should eventually also be applied to other aspects of the blockchain stack to gain (more) confidence in the execution of smart contracts. In particular the implementation of the VM executing the smart contract’s bytecode should receive additional attention. A more recent addition to the blockchain landscape, Tezos[37], is making strides in that regard. The project’s software itself is written in the OCaml, a programming language that has its origin in formal verification research. In addition, the project supports smart contracts in a newly developed language called Michelson that is designed to be amenable to formal verification (see also the next section).

37 https://www.tezos.com

## 4.4 ALTERNATIVE APPROACHES FOR EXPRESSING SMART CONTRACTS

A SMART CONTRACT LANGUAGE IS THE VEHICLE FOR TRANSLATING INTENT INTO CODE THAT IS TO BE EXECUTED ON AND VALIDATED BY THE BLOCKCHAIN NODES. Preferably, such a language should be easy to understand and reason about; a smart contract author should not be surprised by (the semantics of) the language. Unfortunately, the most prolific smart contract language, Ethereum's Solidity, does not fit that description.

Solidity is a statically typed programming language specifically designed for implementing smart contracts on the Ethereum platform. To make Solidity more approachable and familiar for existing web developers, its syntax was designed around ECMAScript (JavaScript). However, over time it has become clear Solidity is a somewhat hastily designed, complex language with many surprising quirks and oddities that make it easy to accidentally introduce security flaws. A sampling of the issues reported elsewhere (from [31], [32]):

- The semantics of operators differ depending on whether the operands are literals or not (e.g., 1/2 is 0.5, but x/y for x=1 and y=2 is 0).
- The order of evaluation is not defined for expressions, which is problematic as the language has value-returning mutating operators like ++.
- Copy is by reference or by value depending on where the operands are stored. This is implicit – the operation looks exactly the same in code.
- The Map data type does not throw an exception on non-existing keys, it just returns the default value.
- Integer overflow and underflow bugs are possible.
- Superficially, Solidity looks like an object oriented language and has a "this" keyword. However, there are security-critical differences between "this.setX()" and "setX()" that can cause wrong results.
- Because the literal 0 type-infers to byte, a for loop like "for (var i = 0; i < a.length; i ++) { a[i] = i; }" will result in an "infinite"[39] loop if a[] has more than 255 elements as i will wrap around to zero.
- Statements allow, but do not require, braces around bodies, which is the cause of a whole class of bugs in C-syntax inspired languages.
- All state is mutable by default.

38 See https://github.com/ethereum/solidity/issues/583.
39 On the Ethereum platform this is not really infinite as execution is limited by the amount of gas provided.

Preferably, a programming language should be easy to understand and reason about; a smart contract author should not be surprised by the language. Unfortunately, Ethereum's Solidity does not fit that description.

As a consequence, people are starting to work on alternatives for Solidity with less surprising semantics, including in the Ethereum community. Viper[40], for example, is an experimental smart contract language that compiles to EVM bytecode and has a python-like syntax that aims for security, simplicity and auditability. The language deliberately does not support certain constructions that were allowed in Solidity but can cause confusion. For example, Viper does not support class inheritance, operator overloading, and recursive calling. In addition, the aforementioned Ethereum formal methods engineer Yoichi Hirai is also experimenting with a new language for the Ethereum platform called Bamboo[41]. This similarly experimental language tries to minimize the chances for surprise by making state transitions explicit and avoiding re-entrancy problems by default. Petterson and Edström [33] also target the Ethereum Virtual Machine, but instead produced a domain-specific language (DSL) inside the existing functional programming language Idris. With Idris' support for dependent types and algebraic side-effects they showed how such an approach can be used to prevent several classes of common errors.

The Ethereum VM (EVM) was for some reason made fairly low level; it uses a stack-based instruction set, more akin to a processor, despite being interpreted. This approach has the downside that such low-level code is hard to reason about, which is why some blockchain projects take a different approach. The Tezos project[42] for example, a public, open source blockchain with a strong focus on governance and correctness, introduced a domain-specific language for writing smart contracts called Michelson[43]. It is a stack-based language (as is the EVM), but in contrast to the EVM bytecode it is strongly typed and it has a number of higher-level operators and data types [34]. Furthermore, the language was specifically designed to facilitate formal verification (see previous section), allowing the users to prove properties of their smart contracts. Another difference is that Michelson contracts are not stored on a blockchain as binary bytecode, but as human-readable text. But even though Michelson code is human-readable, it is still fairly low-level, which is why there are also higher-level languages in development. One of which is Liquidity[44], a smart contract language with an OCaml-like syntax that compiles to Michelson and for which a formal verification framework is under development.

40 https://github.com/ethereum/viper
41 https://github.com/pirapira/bamboo
42 https://www.tezos.com/
43 http://www.michelson-lang.com/
44 http://www.liquidity-lang.org/

**While using an existing, general purpose programming language might make writing smart contract code more approachable initially, there are a number of serious downsides in choosing such an approach.**

The Kadena project, a commercial, permissioned blockchain platform, uses a similar approach and developed a special-purpose smart contract language. Like with Tezos, smart contracts are stored in a (Kadena) blockchain in its human-readable form. Their (open sourced) Pact[45] is a lisp-based, deliberately Turing-incomplete language [35] that favors a declarative, functional approach over complex control-flow, with the aim of making bugs harder to write and easier to spot. The latest version of Pact supports types and allows for the application of formal methods to thoroughly check type correctness [36], but there does not yet appear to be any development on the use of formal methods to verify general high-level invariants.

In the world of permissioned (consortium) blockchains there are several platforms that leverage Ethereum's Solidity and EVM. This includes for example Hyperledger[46] (based on Monax[47]) and JP Morgan's Quorum[48]. Interestingly, many of the blockchain platforms coming out of industrial consortia use or propose to use general purpose languages for smart contracts. For example, in Hyperledger Fabric[49] one needs to use the Go programming language[50] to write smart contracts in (which the platform calls "ChainCode"), with plans to support Java and even JavaScript (node.js) in the future. Similarly, Corda[51] from the financial R3 consortium, uses the Java programming language for smart contract development. While using an existing, general purpose programming language might make writing smart contract code more approachable initially, there are a number of serious downsides in choosing such an approach:

1. It is (too) easy to accidentally write non-deterministic programs that prevent consensus.
2. It is harder to prove high-level properties.
3. There is a real danger of incorporating (existing) non-essential code, producing a larger attack surface.

Finally, we would like to propose to also consider more radically different approaches to expressing smart contracts, preferably those that allow for better communication with domain experts. At TNO, for example, we are working on extending the existing open source Go programming language[52] to generate validated smart contracts from business rules, declaratively expressed [37] in relation algebra. Furthermore, there are various additional avenues to explore for bridging the gap between domain experts and digital smart contract platforms. This includes interactive tooling and visualizations for the exploration of the implications of smart contracts.

45 http://kadena.io/pact/
46 https://www.hyperledger.org/projects/hyperledger-burrow
47 https://monax.io/
48 https://www.jpmorgan.com/global/Quorum
49 https://www.hyperledger.org/projects/fabric
50 https://golang.org/
51 https://www.corda.net/
52 http://ampersandtarski.github.io/

## 4.5 UPGRADE & GOVERNANCE STRATEGIES FOR SMART CONTRACTS

IT IS SOMETIMES USEFUL TO BE ABLE TO MODIFY OR REPLACE SMART CONTRACTS, NOT JUST BECAUSE OF MISTAKES, BUT ALSO DUE TO CHANGING SITUATIONS, NEW INSIGHTS, OR EVEN COURT ORDERS. However, it should be clear for all parties involved under what circumstances smart contract code can be changed. This is something that can and should also be expressed in smart contract code through something like a proxy construction.[53] For example, one could specify that a certain piece of smart contract code can only be changed if 5 out of 9 predefined stewards agree.

Of course, there might still be doubts about the correctness of the governing code itself, though this will be partially mitigated by the fact that the reusability of such governance strategies will allow for the application in a broad range of settings, increasing confidence over time. In addition, the aforementioned formal verification approaches can provide additional assurances on the correctness.

Specifying the governance of smart contracts through other smart contracts is a good way to still keep "a human in the loop".

In a way, such governance of smart contracts through other smart contracts form a middle ground between fully distributed at the one end and centralized at the other. One can also think of it as an escape hatch or a way to still keep "a human in the loop". Which strategies work for which situations is something to be explored and put to the test to reach a set of reusable strategies for multiple smart contract platforms. Further inspiration for this can also be found in traditional contract law. As explored by Marino et al. [38], contract law has developed a well-honed set of tools for altering and undoing contracts, that, while not applicable as-is, appears to be an excellent starting point for upgradeable smart contracts.

Some hands-on advice and examples of upgrade smart contracts for the Ethereum platform is available in for example ConsenSys' smart contract best practices guide [22].

53 In such a construction, calls to a proxy contract are forwarded to another contract, the address of which is stored in the contract. The proxy contract also contains the logic that determines under what circumstances this address can be changed to another address.

## 4.6 LIBRARY OF PATTERNS FOR SMART CONTRACTS

THE UPGRADE STRATEGIES DISCUSSED ABOVE ARE IN FACT ONLY ONE TYPE OF REUSABLE PATTERNS; THERE ARE VARIOUS OTHER PROGRAMMING PATTERNS IN SMART CONTRACTS THAT CAN BE REUSED. This includes for example ways to handle token issuance, ownership of a smart contract, authenticated data providers (oracles), and conditional transfer of funds. An initial first overview of programming patterns found in Ethereum smart contracts is provided in [39]. In terms of reusable code for the Ethereum platform, the OpenZeppelin project provides an open source framework[54] for writing secure smart contracts and includes a number of common contract security patterns.

We believe that a library of battle-tested patterns (and accompanying reference implementations) can help in preventing common mistakes and badly re-invented wheels. As security issues are typically subtle and only surface after a while, a properly documented design pattern can prevent future instances of that same issue. In the world of software engineering, and in particular the area of object-oriented design, design patterns are a well known approach for abstract descriptions of solutions for common problems [40]. The use of design patterns is not without critique, and indeed, care must be taken of course not to *add* complexity and reduce ability to reason about smart contracts by enthusiastically applying patterns.

Furthermore, such patterns for smart contracts should preferably be accompanied by descriptions of formally proven properties such that these patterns can confidently be combined and tuned for specific applications.

Additional developments are needed to support the creation of robust and secure smart contracts that society can depend on.

54 https://github.com/OpenZeppelin/zeppelin-solidity

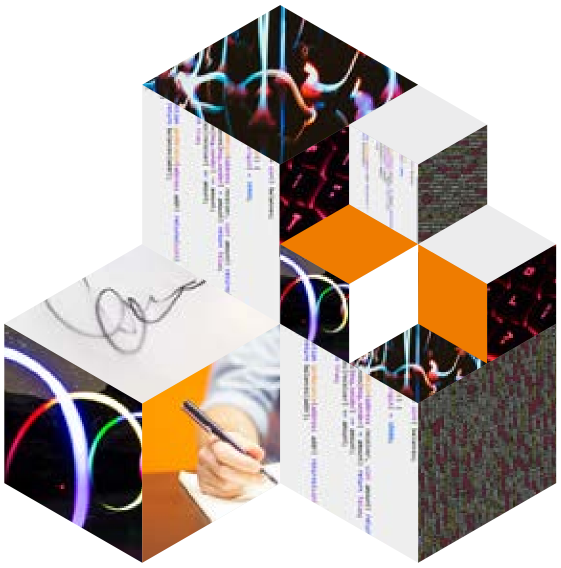
## 5. DISCUSSION AND CONCLUSION

SMART CONTRACTS HAVE THE POTENTIAL TO RADICALLY CHANGE THE WAY WE INTERACT AND TRANSACT WITH EACH OTHER. But before we transfer the control of high-value assets to pieces of code running on a set of distributed (i.e., other peoples') computers, there are number of things to be achieved. Most prominently, smart contracts require a different development process than traditional (web) software development; the hostile environment they run in and the unchangeable nature can make mistakes very costly, as illustrated in this paper. There are already a number of best practices that can help in bridging the gap between intention and code to reduce the chances for (exploitable) bugs. In addition, the application of formal methods has shown great potential in proving high-level (safety) properties. Nonetheless, there is still plenty of room for R&D in this regard.

Unfortunately, the most prevalent smart contract language, Ethereum's Solidity, is embarrassingly unfit for the job of expressing one's intent and expectations. Luckily, there are already some alternatives in development with less surprising semantics, both on the Ethereum platform and on other (new) platforms. But also in this area additional developments are needed for the creation of robust and secure smart contracts.

## ACKNOWLEDGMENTS

We would like to thank Christopher Brewster and Jaco van de Pol for their helpful comments on early drafts of this paper.



# REFERENCES

[1] Veracode, “State of software security 2017,” 2017. [Online]. Available: https://info.veracode.com/report-state-of-software-security.html

[2] Andreas M. Antonopoulos, Mastering bitcoin: Programming the open blockchain, 2nd ed. O’Reilly Media, Inc., 2017.

[3] “Ethereum white paper.” [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[4] Gavin Wood, “Ethereum: A secure decentralised generalised transaction ledger.” [Online]. Available: http://gavwood.com/paper.pdf

[5] Nick Szabo, “Smart contracts: Building blocks for digital markets,” 1996. [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html

[6] Amy Castor, “One of ethereum’s earliest smart contract languages is headed for retirement,” August-2017. [Online]. Available: https://www.coindesk.com/one-of-ethereums-earliest-smart-contract-languages-is-headed-for-retirement/

[7] Phil Daian, “Analysis of the dao exploit,” June-2016. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[8] Lawrence Lessig, Code 2.0, 2nd ed. Paramount, CA: CreateSpace, 2009.

[9] Matthew Leising, “The ether thief,” June-2017. [Online]. Available: https://www.bloomberg.com/features/2017-the-ether-thief/

[10] Santagio Palladino, “The parity wallet hack explained,” July-2017. [Online]. Available: https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[11] Alyssa Hertig, “Ethereum client bug freezes user funds as fallout remains uncertain,” November-2017. [Online]. Available: https://www.coindesk.com/ethereum-client-bug-freezes-user-funds-fallout-remains-uncertain/

[12] Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi, “Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab,” IACR Cryptology ePrint Archive, vol. 2015, p. 460, 2015.

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor, “Making smart contracts smarter,” in ACM conference on computer and communications security, 2016, pp. 254–269.

[14] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli, “A survey of attacks on ethereum smart contracts sok,” in Proceedings of the 6th international conference on principles of security and trust - volume 10204, 2017, pp. 164–186 [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[15] Vlad Zamfir Dino Mark and Emin Gün Sirer, “A call for a temporary moratorium on the dao,” May-2016. [Online]. Available: http://hackingdistributed.com/2016/05/27/dao-call-for-moratorium/

[16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou, “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts,” in IEEE symposium on security and privacy, SP 2016, san jose, ca, usa, may 22-26, 2016, 2016, pp. 839–858 [Online]. Available: https://doi.org/10.1109/SP.2016.55

[17] Nancy G. Leveson, “The role of software in spacecraft accidents,” AIAA Journal of Spacecraft and Rockets, vol. 41, pp. 564–575, 2004.

[18] Vitalik Buterin, “Thinking about smart contract security,” June-2016. [Online]. Available: https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/

[19] Solidity Smart contract Security best practices, “Nikhil mohan,” August-2017. [Online]. Available: https://lightrains.com/blogs/smart-contract-best-practices-solidity

[20] Manuel Araoz, “Onward with ethereum smart contract security,” August-2016. [Online]. Available: https://blog.zeppelin.solutions/onward-with-ethereum-smart-contract-security-97a827e47702

[21] Ethereum Project, “Solidity - security considerations.” [Online]. Available: http://solidity.readthedocs.io/en/develop/security-considerations.html

[22] ConsenSys, “Ethereum contract security techniques and tips,” September-2017. [Online]. Available: https://github.com/ConsenSys/smart-contract-best-practices

[23] Eric Rafaloff, “Reviewing ethereum smart contracts,” September-2017. [Online]. Available: https://blog.gdssecurity.com/labs/2017/9/27/reviewing-ethereum-smart-contracts.html

[24] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu, “KEVM: A complete semantics of the ethereum virtual machine,” August 2017 [Online]. Available: http://hdl.handle.net/2142/97207

[25] Christian Reitwiessner, “Dev update: Formal methods,” September-2016. [Online]. Available: https://blog.ethereum.org/2016/09/01/formal-methods-roadmap/

[26] Ilya Sergey and Aquinas Hobor, “A concurrent perspective on smart contracts,” CoRR, vol. abs/1702.05511, 2017.

[27] Christian Reitwiessner, “Formal verification for solidity contracts,” October-2015. [Online]. Available: https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts

[28] Jean-Christophe Filliâtre and Andrei Paskevich, “Why3 – Where Programs Meet Provers,” in ESOP’13 22nd European Symposium on Programming, 2013, vol. 7792 [Online]. Available: https://hal.inria.fr/hal-00789533

[29] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin, “Formal verification of smart contracts: Short paper,” in Proceedings of the 2016 acm workshop on programming languages and analysis for security, 2016, pp. 91–96 [Online]. Available: http://doi.acm.org/10.1145/2993600.2993611

[30] Yoichi Hirai, “Formal verification of deed contract in ethereum name service,” November-2016. [Online]. Available: https://yoichihirai.com/deed.pdf

[31] peoplewindow, “HackerNews comment on “underhanded solidity coding contest”,” July-2017. [Online]. Available: https://news.ycombinator.com/item?id=14691212

[32] “HackerNews comment on “153k ether stolen in parity multi-sig attack”,” July-2017. [Online]. Available: https://news.ycombinator.com/item?id=14810008

[33] Jack Pettersson and Robert Edström, “Safer smart contracts through type-driven development,” Master’s thesis, Institutionen för data- och informationsteknik (Chalmers), Chalmers tekniska högskola, 2016.

[34] Tezos, September-2017. [Online]. Available: https://github.com/tezos/tezos/blob/master/src/proto/alpha/docs/language.md

[35] Stuart Popejoy, “The pact smart-contract language,” June-2017. [Online]. Available: http://kadena.io/docs/Kadena-PactWhitepaper.pdf

[36] Stuart Popejoy, “Types (and type inference) in pact,” January-2017. [Online]. Available: http://kadena.io/blog/

[37] Gerard Michels, Sebastiaan Joosten, Jaap van der Woude, and Stef Joosten, “Ampersand,” in Relational and algebraic methods in computer science: 12th international conference, ramics 2011, rotterdam, the netherlands, may 30 – june 3, 2011. Proceedings, H. de Swart, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 280–293 [Online]. Available: https://doi.org/10.1007/978-3-642-21070-9_21
[38] Bill Marino and Ari Juels, “Setting standards for altering and undoing smart contracts,” in Rule technologies. Research, tools, and applications: 10th international symposium, ruleml 2016, stony brook, ny, usa, july 6-9, 2016. Proceedings, J. J. Alferes, L. Bertossi, G. Governatori, P. Fodor, and D. Roman, Eds. Cham: Springer International Publishing, 2016, pp. 151–166 [Online]. Available: https://doi.org/10.1007/978-3-319-42019-6_10
[39] Massimo Bartoletti and Livio Pompianu, “An empirical analysis of smart contracts: Platforms, applications, and design patterns,” CoRR, vol. abs/1703.06322, 2017.
[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design patterns: Elements of reusable object-oriented software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

*januari 2018*

CONTACT
**Maarten Everts**
Unit ICT – Cyber Security and Robustness
Locatie Groningen
maarten.everts@tno.nl
088 866 31 90

TNO innovation for life

TNO.NL

17-9483