SMT-based Formal Verification for Smart Contracts: the State-of-the-Art

Tobias Rothmann
Technical University of Munich
Munich, Germany
tobias.rothmann@tum.de

***Abstract*—Smart Contracts are programs that run on a blockchain. One of their main purposes on public blockchains like Ethereum is asset management. Since their byte code, if not their source code, is public their correctness is of uttermost importance. To formally prove their correctness, while keeping the workload as low as possible, SMT-based formal verification tools came up in recent years, that prove given specifications automatically with SMT-solvers. Although these tools differ in their capabilities and limitations, no resource points out and categorizes these differences. This paper attempts to picture the state-of-the-art by categorizing and comparing all existing tools. The tools are compared with regard to completeness, in the sense of limits and the complexity of specifications that can be proved, and with regard to optimization, in the sense of preprocessing of specifications and smart contract code to allow for faster verification through SMT-solvers. Thus giving an overview of how formal verification with certain tools is to judge, which tool to use to formally verify one's project and most importantly which research challenges remain open.**

I. Introduction

Blockchain, a buzzword for many. However, over the years the market for blockchain technology has evolved and come further from Bitcoin's approach of solely transacting digital cryptocurrencies, with Ethereum [1], being the first blockchain to implement executable smart contracts, which are programs that are run on a blockchain.

One of the main purposes of smart contracts is the asset management of cryptocurrencies. Over the past years, the DeFi (decentralized finance) industry has evolved, which uses smart contracts for financial purposes. The amount of money that is locked in DeFi smart contracts as of writing this paper, is around 45 billion USD [40]. With single platforms like AAVE, having cryptocurrencies worth 6 billion USD locked in their smart contracts alone [40]. Hence the security of smart contracts is of paramount importance, as one bug could lead to losses in the billions. The past has shown how important the security of smart contracts is, with hacks leading to multi-million USD losses already [36]. Audits and testing are already part of the best practices for smart contract development [41], however, they can take one only so far, as they can, to follow the words of Edsger W. Dijkstra, "only show the presence of bugs but not their absence". Formal methods, on the other hand, can mathematically prove properties for smart contracts and have already been utilized to make smart contracts more secure [38]. Though most of the approaches so far have been focusing on automatic static-analysis tools, that are overapproximate for the sake of being automatic or only solve certain problems [38], or interactive theorem prover approaches that are complete, but require expertise in them and proving techniques in general [38]. However, besides that, there are also SMT-based approaches, which utilize the many decades of research that was put into making SMT-solvers as efficient as possible [35]. SMT-solvers try to solve undecidable problems in first-order logic (FOL), by utilizing SAT solvers and domain-specific theories, thus they have an extensive application catalog and hence a broad research base that has been developing them for decades [4], [35]. SMT-based approaches can utilize the expressiveness of SMT-solvers and reduce a program with user-given specifications to a FOL expression, that they can give to an SMT-solver to solve automatically. Hence SMT-based approaches can combine the completeness of interactive theorem provers with the automaticity of overapproximate static analysis tools, with some exceptions of course, as formal verification is undecidable due to the Halting problem.

In this paper, we investigate the current state-of-the-art for SMT-based formal verification, specifically for smart contracts. Therefore we consider all SMT-based formal verification tools which aim for a complete approach, namely the SMT-Checker [14], solc-verify [11], the Certora prover [18] and the Move prover [15]. The broad research question for this paper is:
- What is the state-of-the-art for SMT-based formal verification tools for smart contracts?
With the goals of giving an overview of the possibilities and capabilities of the field to researchers and thus fostering further research in the field, and also making the topic accessible to people outside the field.

To answer the research question the tools are evaluated and later compared with regard to completeness, in the sense of completeness-limits of the tools(e.g. loop-verification) and expressiveness of specifications they allow, and optimizations they implement to facilitate faster and better results. For Completeness, each tool is evaluated in 4 categories:
1. whether a tool can verify loops for all possible executions (i.e. is unbounded) or only for a bounded set of repetitions of

the loop-body (i.e. is bound).
2. whether the verification is performed on source-code or bytecode, as source-code verification, might be invalid for bytecode due to compiler bugs.
3. whether the specifications are expressive enough to cover the whole contract (i.e. contract scope) or are limited to single functions (function scope).
4. whether the specifications are expressive enough to allow quantification over variables or not.
For Optimizations, the tools were inspected and common approaches were unified, such that an overview of shared approaches can be given. Thus every tool is categorized on if they implement a certain optimization or not. Common approaches that were unified are:
1. non-aliasing memory model, tools that use a non-aliasing memory eliminate the need for SMT-solvers to reason about aliasing and facilitate simplifications such as replacing references with values (reference elimination).
2. intermediate language representation, intermediate representations in a different language can help to optimize SMT-solving as the intermediate language can be better optimized for SMT-solving than a direct translation of program code into SMT theory.
3. simplification, simplifying the code or intermediate language before translating it to the SMT-solver. Thus speeding up the solver.

II. Background

To understand how a smart contract can be automatically verified using an SMT-solver, given only specifications, the essential definitions and concepts of SMT, specifications, smart contracts, and concepts surrounding them need to be defined and explained. This section aims to do exactly that, such that the reader is capable of understanding the remaining paper.

A. Blockchain

The key principles of blockchain are decentralization, trustless interaction/transaction of assets, and finally, which also means irreversibility.
Every interaction with a blockchain is realized in a transaction, which is processed by a decentralized network and then build into a block, which is cryptographically dependent on all previous blocks and therefore irreversible. A permissionless blockchain is called a public blockchain, meaning everyone can send transactions to the network and access its transaction history.

B. Smart Contract

Smart Contracts are programs that run on a blockchain.
Some blockchains, like Ethereum, add a virtual machine (VM) on top of the transaction layer, which allows for the execution of programs, called smart contracts [1]. Their compiled bytecode, which can be directly run by the VM, is uploaded via a transaction and therefore publicly inspectable on the blockchain. Smart contracts are invoked through transactions and can only operate on data stored on the blockchain and in the transaction. Smart contracts can invoke and interact with other smart contracts.

The Porvers in this paper work only on two smart-contract languages, Solidity [25] and Move [28]. Thus we explain them both.

C. Solidity

Solidity [25] is the most popular smart contract language [24] and was introduced in 2015 [25]. Solidity is an Ethereum Virtual Machine (EVM) based programming language, hence it compiles to EVM-Bytecode [26]. It is a contract-oriented programming language, which is derived from object-oriented programming languages and thus smart contracts in Solidity can inherit from other smart contracts and have their logic capsulated in functions, that must be individually invoked [26]. Besides the typical object-oriented and programming language features, Solidity has some specialties for the interaction with the blockchain and specifically the EVM [26].

1) reverting: Whenever an error occurs in the program, the EVM ”reverts”, meaning all changes since the transaction that triggered the initial computation are reverted to the state before the transaction [1]. Effectively, turning erroneous program paths into NOPs (No Operation Instruction). Solidity even supports constructs that trigger errors, such that the whole computation is reverted, namely ”require”-statements [26]. Require statements take a bool expression as an argument and revert if the expression evaluates to false [26].

2) gas-consumption: For the execution of every Opcode must be paid, via the blockchain native cryptocurrency [1]. Even if a transaction is reverted, the consumed gas to get to the error must be paid [26].

3) memory model: Since the EVM is virtual, there is no real hardware like RAM-memory assumed. Instead, two kinds of abstract memory can be accessed throughout the contract, namely memory and storage [26].
1. memory: memory is temporary and will be deleted after the execution of the function is done [26].
2. storage: storage is permanently stored on the blockchain. Furthermore, the state of a smart contract is equivalent to its storage [26].

D. Move

Move [28] is, compared to Solidity [25], a younger language, proposed in 2019 and first implemented and used in the Aptos blockchain in 2022 [32].
Move is Rust-based [33]. It implements a Borrow-Checker, similar to Rust, which ensures that there is either exactly one mutable reference or arbitrary immutable references to a certain space in memory [33]. Move also implements most of the common Rust language features, like Structs, Vectores, Ownership of memory and generic functions [33].
Move supports two types of executable programs:

1) 1. Scripts: Scripts are stateless, meaning they do not have persistent memory on the blockchain and thus are only used to define simple functions and operations. They can only have one function, named main [33].

2) 2. Modules: Modules are the common way for defining smart contracts in Move [33]. They are similar to Solidity smart contracts meaning, they have persistent memory on the blockchain and can import other smart contracts [33], however, they cannot inherit, as Move is no object/contract-oriented language, but Rust-based. Hence Move imports from other Modules behave like Rust imports, meaning they import only functions and types. The only entry to modules are as "public" defined functions [33]. Modules' internal memory, although persistent, is not directly accessible from outside the module [33].

Move just like Solidity, reverts if an error occurs and consumes gas for every bytecode executed, independent of the result (i.e. revert or not) [28], [33].

E. Specification

Specifications are formalized semantic properties of a program. Common assertions ins programming languages are a type of specification that can express simple-semantic properties for variables by comparison. Another stronger type are quantified specifications, which have the capability of quantifying over variables (e.g. with forall and exists quantifiers). Specifications are usually not bound to programming languages expressions, but can use more expressive logics like first-order logic (FOL) or higher-order logic (HOL).

F. Hoare-Logic

Hoare-Logic is used for reasoning about the semantics of imperative programs. Hoare triples are constructed with a precondition P, a postcondition Q, and a program c, such that

> "If P is true before the execution of c then c terminates and Q is true afterwards." [2]

The formula is written as $\{P\}c\{Q\}$ [2].

Hoare triples can be reduced to logical formulas, such that the Hoare triple is valid if and only if the logical formula is provable (a tautology) [2]. Such a logical formula is called a verification condition and the function that generates it verification condition generator (VCG) [2].

Preconditions can be strengthened and Postconditions weakened: Given $P' \implies P$,P' is stronger than P, $Q \implies Q'$, Q' is weaker than Q, and $\{P\}c\{Q\}$ is valid follows that $\{P'\}c\{Q'\}$ is also valid [2].

$$\frac{P' \implies P \ \{P\}c\{Q\} \ Q \implies Q'}{\{P'\}c\{Q'\}}$$

[2]

Hoare triples are defined over each command of a language [2]. Therefore a program c that consists of commands $c_1 \dots c_n$ can be simplified to multiple intermediate Hoare triples for each command:

$$\begin{aligned} \{P\}c\{Q\} &= \{P\}c_1 \dots c_n\{Q\} \\ &= \{P\}c_1\{Q_1/P_2\} \dots \{Q_{n-1}/P_n\}c_n\{Q\} \end{aligned} \tag{1}$$

[2]

Hoare logic also provides a rule for loops, making them provable for all possible state transitions in the program [2]. Hence a sound and complete proof scheme for loops. The rule says: given the Hoare triple $\{P \wedge b\}c\{P\}$, the Hoare-triple $\{P\}$ WHILE b DO c$\{P \wedge \neg b\}$ is valid [2]:

$$\frac{\{P \wedge b\}c\{P\}}{\{P\}\text{WHILE b DO c}\{P \wedge \neg b\}}$$

[2]

G. Inductive Verification of Smart Contracts

Inductive Verification is about showing that a property holds for all states the smart contract can take. Such a property, that should always hold, is called a invariant. The invariant is, similar to textbook induction, shown via a base case and possible multiple induction steps. The base case is the initialization of the smart contract (e.g. the property must hold after the constructor in Solidity and after the initialization of global memory in Move). The induction step is to show for every public function that if the invariant is assumed before the function, it holds after the function. Since state changes can only be made through the invocation of public functions, the induction is sound and complete. The detailed proof can be found in the Certora docs [19].

H. Symbolic reasoning

Symbolic reasoning abstracts concrete values into symbolic ones [34]. A prominent example of the application of symbolic reasoning is reasoning about integers, where integers are abstracted into intervals. Thus in the reasoning, some arbitrary integer in a program is replaced by an interval of all possible values that the arbitrary integer could have taken.

I. Formal Verification

We say a program is formally verified if we have a proof that shows that the given specifications for this program hold (e.g. one has a valid hoare-triple for the program with an empty precondition and a post-condition that implies the specification).

J. SMT-solver

Satisfiability Modulo Theories (SMT) solvers are automated theorem provers [35]. SMT-solver are based on SAT solvers and extend them to check the satisfiability of first-order logic (FOL) expressions, though that is not complete, as satisfiability for FOL is undecidable [35]. They are especially useful in formal verification, as verification conditions generated by VCGs, reduced to FOL, can be given to SMT-solvers to prove. Although that is not guaranteed to work, as formal verification is undecidable since the Halting problem is undecidable

(therefore it also makes sense that FOL is undecidable) [4].
Though FOL, in general, is undecidable, certain domains of FOL can be solved, due to domain-specific optimization and properties [35]. There is a "long and still growing list" [4] of theories that have "decision procedures", which are algorithms that determine the satisfiability of a specific domain, hence making certain domains decidable [4]. These domains include finite bit-vectors, lists, trees, tuples, hash tables and more [4], [35]. Furthermore, domain-specific theories with decision procedures can be combined to solve complex mixed-theory problems (via e.g. the Nelson-Oppen Procedure), like lists with bit-vectors as 64bit-integer [4], [35].
Hence there exist decidable theories within SMT-solving that solve specific domains important for program verification.
Domains with decision procedures are mostly quantifier-free [35], meaning they disallow quantifiers. However, introducing predicate symbols that are not in the domain theory also introduces quantifiers [35]. Furthermore, program verification formulas use quantifiers to express properties e.g. about elements in data structures [35].
Although SMT-solvers implement techniques for quantifier-elimination, which is undecidable [35], there are clear implications from this for SMT-based program verification:
1. Tailoring the verification conditions to specific domains that are decidable (i.e. have a decision procedure) should be a goal of every SMT-based approach.
2. Quantifiers will hinder SMT reasoning and thus should be avoided where possible.

K. Boogie & the Boogie IVL

Boogie is a tool that is optimized for SMT-based verification of programs [7], [22]. It takes programs annotated with specifications defined in the Boogie Intermediate Verification Language (IVL) and reduces them to an SMT model that is off-loaded to SMT-solver for solving [22].
The IVL is not executable but optimized for the reduction to SMT theories with decision procedures [22]. It supports common programming language features, such as loops, if-then-else-conditions, local and global variables, functions, function calls and more (the reader is referred to [22] for a complete listing). Furthermore, the IVL supports Hoare-style pre and post-conditions, as well as loop-invariants [22]. The IVL is closely designed to SMT theory and implements arrays, integers and datatypes according to the SMT theories with decision procedures [22]. Due to that and other optimizations, listed in [22], SMT-based solving of IVL programs is significantly more efficient than the solving of common translation approaches of programming languages into SMT theory [7], [22].
Boogie is the verifier around the IVL, that reduces the IVL programs into SMT theory (i.e. VCG), calls the SMT solvers on the reduction result and returns the verification or a violation [7].

L. SMT-solving and smart contracts

This section aims to give background on why SMT-solving makes sense for the formal verification of smart contracts. There are 3 main reasons for that:

1) financial incentives: Smart contracts are designed to manage financial assets [1] and thus a bug is of great consequence, possibly leading to multi-million-dollar losses [36]. Therefore formal verification is of uttermost importance.

2) convenience: SMT-solvers solely need the code and specifications, thus enabling developers, who do not have expertise in formal verification proof techniques and interactive theorem provers, to formally verify their contracts. Furthermore, paper proofs and even proofs with interactive theorem provers take time and concentration, while SMT-solvers are automatic and faster if they can prove a statement. Hence adoption is much more likely.

3) technical prerequisites: SMT-solver have been developed for a few decades now [4], [35], thus are technically advanced and have an established research base. Smart contracts are incentivized to be as short as possible and only implement the program logic, as every command costs money to execute. Resulting in, compared to conventional software, smaller VCs, which in consequence, reduces the complexity and time needed for the SMT-solving. Furthermore, Smart Contracts are a closed system, without concurrent processes [1], [32], which need quantifiers, and thus easier to reason about compared to conventional software.
In summary, smart contracts have high incentives for formal verification, SMT-solvers are faster and easier to use for developers, and smart contracts produce easier VCs than conventional software, hence are easier to proof for SMT-solvers.

M. Applied SMT-based approaches

The approaches covered in this paper are:

1) scol-verify: Solc-verify is an add-on to the Solidity compiler. It takes specifications in Solidity expressions, embedded in comments, and translates the code with specifications into Boogie, which of-loads the verification to SMT-solvers [11]. Special optimizations of the approach are:
1. it abstracts the Solidity memory model, in the translation, into a more SMT-friendly memory model in Boogie and
2. it provides a modular arithmetic mode in Boogie to increase efficiency for Solidity's 256-bit integers [11].

2) SMT-Checker: SMT-Checker was the first SMT-based formal verification approach for smart contracts, published in 2018 [14]. SMT-Checker came with the standard Solidity compiler solc [26], but has since been replaced by it's successor's SolCMC [12] and Solicitous [13], which both are Constrained Horn Clause (CHC) based and are discussed in the related work section. Though SMT-Checker is an early project and little refined, it is part of this paper because of its pioneering work in this field and for completeness reasons.

3) The Move Prover: The Move Prover (MVP) is integrated into the Move language, which was developed together with the MVP, which thus has the tightest language integration of

this paper [15]. The MVP takes Move-contracts with annotated specifications and compiles the Move-sourcecode to bytecode while extracting the specifications [15]. The compiled bytecode-level model is simplified, using optimizations such as reference elimination and monomorphization, and translated into Boogie's IVL [16], [22]. Boogie translates the IVL code to an SMT model, which then is given to SMT solvers to solve, namely CVC5 and Z3 [7]–[9], [16]. The MVP maps back the result of the SMT-solvers, which is either the verification or a counterexample, to the source code.

4) The Certora prover: The Certora prover is the only corporate and therefore not open-source prover in this paper. However, it supports a lot of optimizations and has a unique approach, which is why it has a place in this paper. It takes Specifications in a separate file in their specification language CVL (Certora Verification Language) and translates them into the intermediate language TAC (three address code) [17]. The Sourcecode is compiled into bytecode, which is also translated to TAC [17]. After simplification of the combined TAC model, the from TAC generated verification conditions are offloaded to SMT-solvers which report satisfiability (i.e. verification) or give a counter-example [17].

III. Systemization and Taxonomy

The tools are compared with regard to two broader categories, completeness and optimization. Both categories have multiple subcategories which evaluate subaspects of them.

A. Completeness

Completeness is a measure of the limits and complexity of specifications the tools allow. This section will be more useful to people who want to use the tools or evaluate formal verification with certain tools. For limits, boundedness with regard to loop verification will be assessed. The complexity of specifications is split up into three subcategories, Bytecode/Sourcecode level, Contract/Function scope, and quantified specifiactions. Each subcategory presents two options of which one is fulfilled (by mutual exclusion), except for one exception where also both options can be fulfilled.

1) unbounded/bounded: The tools are divided into unbounded and bounded tools, with regard to how they handle loop verification. Unbounded tools refer to tools that prove verification conditions of loops sound and complete, with regard to every possible execution. Hoare logic with loop invariants, therefore, is unbounded. Bounded tools refer to tools that prove conditions only over a bounded set of possible executions. This occurs for example in loop-unrolling approaches.

2) Bytecode/Sourcecode level: The tools are divided into bytecode level and sourcecode level with regard to what level the formal verification is performed on. Blockchain VMs execute just the bytecode, therefore only formal verification over the semantics of the bytecode can express security guarantees. Sourcecode semantics are designed to exactly match the translation into bytecode semantics, however as long as the compiler, who translates the source code into bytecode, is not proven to do that, there is no guarantee. Hence formal verification of source code might be invalid for the bytecode due to compiler bugs. As of writing this paper, I am not aware of any verified compiler for either one of the two languages the tools examined cover, namely Solidity and Move. However, bugs in a compiler for Solidity have been found [5].
Despite the down-side source code level formal verification has, it also has an up-side. All tools examined in this paper use specifications defined on the source code level, which do not have to be translated to the bytecode level for source code formal verification.

3) Contract/Function scope: The tools are divided into contract and function scope with regard to what specifications can be expressed over. Function scope tools can only express specifications within functions, while contract scope tools can express specifications contract-wide over contract variables and functions, hence contract scope formal verification tools are more expressive. This advantage is especially desirable when expressing global specifications not bound to a function. For example that if the total balance of the smart contract changes, the balance of one address stored in an internal mapping must change for the same amount.

4) quantified specifiactions: The tools are divided into tools that allow quantified specifications and tools that do not. Quantified specifications are specifications that allow quantifiers to be used, i.e. to quantify over variables with forall and exists quantifiers. Thus making the specifications more expressive. Although SMT-solvers have their difficulties with quantifier [23], they allow for precise and human-readable specifications, which is why they are important to consider.

B. Optimization

Optimization is a comparison of techniques used for prover acceleration. This section will be more useful for people who want to understand the technical details. Each subcategory represents one optimization approach therefore, a specific tool either makes use of it or not.

1) non-aliasing memory (abstraction): Aliasing is having multiple different pointers, pointing to the same space in memory, which leads to additional complexity in SMT solving [10]. The memory function is not injective for aliased-memory - spaces and thus no trivial simplification can be made, instead a quantification over the preimage of a single slot in memory or similar complexities have to be made. Non-aliasing disallows aliases and thereby enforces bijectivity of the memory function therefore, it reduces complexity with regard to reasoning about memory changes from other references and adds simplification rules, namely Reference Elimination, which allows inserting direct values instead of references, as these cannot change.

2) intermediate language representation: Instead of trying to reduce code and specifications directly to an SMT-level-description like smt-lib [6], code and specifications can be translated into a smt-friendly language, that is optimized for SMT-reasoning and still preserves programming language style, instead of the SMT typical FOL style. A prominent

example is Boogie [7] with its IVL [22], that preserves imperative programming syntax, while being optimized for SMT theories with decision procedures [22].

3) preprocessing wrt. VC simplification: Preprocessing with regard to VC simplification presents a broad category, that incorporates every kind of simplification of code and specifications that lead to simpler verification conditions and thus to shorter reasoning time. While there is no common pattern in the preprocessing techniques used by the individual tools, this section remains important to understand how researchers are fastening their tools. Simplifications in this section include among others replacing references with values via reference elimination, static analysis of code to infer invariants relevant to the specifications, and inferring more precise types of functions via monomorphization.

IV. Application

The application of taxonomy and systemization is divided into three parts for every tool:
1. Completeness
The Completeness properties of each tool are examined.
2. Architecture
To give a brief overview of the approach of the tool, the basic architecture is discussed, on top of which the evaluation of Optimizations follows.
3. Optimization
The Optimizations of each tool are systemized. Additionally, specialties of optimizations are covered more deeply.

A. solc-verify

1) Completeness: Solc-verify is a Sourcecode level formal verification tool, as it performs the reasoning on the Solidity source code [11]. All specifications are given as Solidity expressions with some additional constructs such as quantifiers (forall and exists) and functions over data structures such as ”sum” [27]. There are 3 types of specifications in solc-verify, which are all annotated in a comment above the function/contract/loop they are referring to:
1. Hoare-style pre and post-conditions
Preconditions are assumed at the beginning of the function and post-conditions are to be shown to hold after the function, via SMT solving [11].
2. Hoare-style Loop invariants
Loop invariants have to hold at the beginning of the loop and after each iteration [11]. They are proved inductively by the SMT solvers, thus achieving unbounded formal verification [11]. **3. Contract invariants**
Contract invariants are defined as pre and post-conditions for every function, hence they are assumed at the beginning of every function and have to hold after every function [11]. Additionally, they have to hold after the constructor, without being assumed before it [11]. Therefore, specifications about contract variables can be expressed and contract invariants are contract scope.

2) Architecture: The architecture of solc-verify is also depicted in Fig. 1 for a quick overview. Solc-verify has an extended Solidity compiler, which extracts an Abstract Syntax Tree (AST) of the source code and the specifications for the formal verification of the comments [11]. The AST and the specifications are then translated into Boogie's IVL, which is optimized for SMT reasoning [22] and on a similar high level as Solidity, thus making the translation easier [11]. Boogie translates the IVL to an SMT model, which is then given to the SMT solvers to solve [7], [11], [22]. If the SMT-solvers give back a counter-example for an unsatisfiable SMT-model, solc-verify maps the counter-example back to the source code and outputs line-numbers and function names that violate the specifiactions [11].

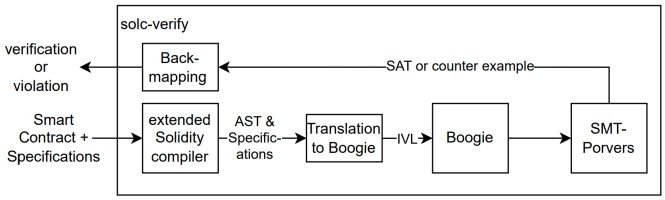

Fig. 1. solc-verify architecture

3) Optimization: Solc-verify translates the AST to Boogie, as it already uses reference elimination to resolve references to memory and types, therefore, simplifying the code [11]. The translation into Boogie's IVL optimizes the SMT-solving for two reasons, the first being the IVL is optimized for SMT-solving [22] and the second being, that the translation of Solidity into Boogie is for the most part simple [10]. The contracts are translated into one Boogie program, where each contract's variables are accessed via a mapping over contract addresses [11]. Most Solidity language blocks can be directly or trivially (e.g. for loop to while loop) translated to corresponding language blocks in Boogie [11]. Solidity specialities, such as fallback functions and transactions are additionally added in Boogie [11]. Apart from the mere translation into Boogie, solc-verify furthermore implements three special optimizations, to speed up the SMT-reasoning:
1. memory-abstraction in SMT-theory
Solc-verify implements a symbolic memory abstraction into SMT theory, which makes use of SMT-theories with decision procedures. The memory-abstraction is based on a complete semantic of Solidity features in SMT theory, which is described in detail in [10]. The abstraction divides Solidity's features (i.e. data structures, types, structs etc.) into storage and memory-based subcategories [11]. This eases reasoning, as the value semantics of storage erases the necessity for reasoning about non-aliasing [10]. Most language features are abstracted into SMT-theory without the need for quantifiers, further easing SMT-solving [10], [35]. Data structures, such as arrays and maps are abstracted into SMT-theory arrays

and inductive datatypes, which have decision procedures [10], [35]. All value types, except for booleans and including addresses, are mapped to SMT-integers, while reference-types (i.e. types referencing other types) are based on SMT arrays and inductive data types, which do not require quantifiers [10]. The non-aliasing property of pointers to storage and memory is a special optimization in itself and thus is discussed in the next paragraph. [10]

2. non-aliasing memory-abstraction

Solc-verify implements a symbolic memory abstraction in SMT-theory, which among others aims to ensure non-aliasing properties as much as possible [10]. Solidity's storage is by design non-aliasing, which solc-verify uses and extends to storage pointers [10]. They use the fact that Solidity's storage can be represented as a finite-depth tree and formalize storage pointers as SMT-arrays, that represent paths in the tree. Thus ensuring the non-aliasing of storage pointers. They use two functions, pack and unpack, which create the path encoding of the storage tree (pack) and follow back the path in the storage tree to "dereference" the pointer (unpack). Furthermore, this abstraction allows for SMT reasoning about the storage pointers without quantifiers, which eases SMT-solving [10], [35].

3. modular arithmetic mode

SMT-theory provides two default ways of dealing with integers, the *mathematical integer mode*, which does not support overflows, but is efficient to solve for SMT-solvers, and the *bitvector mode*, which supports semantics for overflows, but due to Solidity's large 256-bit integers is not efficient for SMT-solving [11]. Solc-verify implements a third way, the *modular arithmetic mode*, which internally works with mathematical integers, but covers overflow semantics via range assertions and implements wrap-around semantics, thus enabling efficient reasoning about integers without a compromise on overflow and wrap-around problems [11].

B. SMT-Checker

1) Completeness: The SMT-Checker performs its verification on the Solidity source code and thus is a sourcecode-level formal verification tool [14]. It takes Solidity's "requires" as pre-conditions and "asserts" as post-conditions for specifications [14]. Additional expressiveness through constructs like quantifiers is not provided. Hence the SMT-Checker is only a function scope tool [14]. Loop invariants are not supported and not inferred, instead, the loop is unrolled exactly one time, which is equivalent to leaving the loop construct away and just inserting the loop body instead of the whole loop [14]. Hence the SMT-Checker is bounded.

2) Architecture: The Architecture of SMT-Checker is depicted in Fig.2 for a quick overview. The SMT-Checker extracts the AST and the specifications of the program and directly translates them into SMT constraints [14]. No Hoare-style reasoning is applied in the translation, instead, pre and post-conditions are constrained together with the program in one SMT model for the provers [14]. "Control flow", "type constraints", "variable assignments", "branch conditions", and the "Verification Target" [14], are all constrained separately to form one SMT-model of program semantics and specifications [14]. The translation is for the most part straightforward (i.e. branch conditions to path exploration, verification target to assertions and so on, the reader is referred to [14] for a comprehensive overview), except for "variable assignments" [14]. "Variable assignments" are translated into new variables that are assigned exactly one time. [14] The resulting SMT model is then given to the SMT-solvers Z3 [8] and CVC4 [30] via their C++ APIs. The SMT-solvers return either the verification result or a violation, which is mapped back to the source code by the SMT-Checker [14], [25].

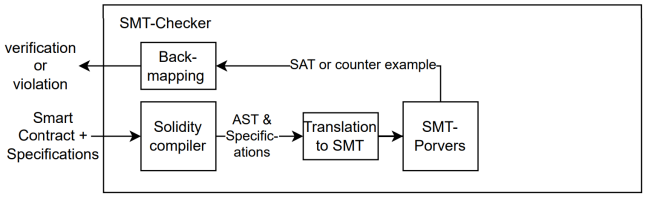

Fig. 2. SMT-Checker architecture

3) Optimization: The SMT-Checker works on the AST, as it already eliminates references to memory where possible [11], [14]. Apart from that it solely translates the AST from the Solidity source-code into SMT-constraints and thus does not incorporate other optimizations such as a possibly non-aliasing memory abstraction or any other simplifications [14]. Though one could argue, that translating the AST directly into an SMT-model bypasses the extra step of translation into another language and thus could be an optimization, this is not the case for the SMT-Checkers implementation. The Boogie language for example is heavily optmized for SMT-solving and easy to translate into from Solidity [10], [22]. Whereas the SMT-Checker provides a simple translation with overheads like new variables for every assignment and introduces quantifiers for connecting control-flow, branch-conditions etc. [14]. Hence its translation process is not as optimized for SMT-solving as a translation into Boogie's IVL would have been [14], [22]. This is also seen in performance examples [11].

C. The Move Prover

1) Completeness: The MVP is a Bytecodelevel formal verification tool, as it performs the reasoning on compiled Bytecode of Move-contracts [15]. Specifications, though in the contracts included, are formulated in a separate specification language, the Move Specification Language (MSL) [15]. The MSL is syntactically similar to Move and extends the expressiveness of Move, meaning it can express everything that Move can plus some extra constructs, with one exception [28]. Functions that return a "&mut T" type cannot be expressed in the MSL, but in Move, though the authors claim that, that this construct is not commonly used in Move development [28]. The extensions of MLS to Move include among others:

1. Quantfiers
MLS supports forall and exists quantifiers over all types [28].
2. built-in constants and functions
MLS features constants like MAX and MIN, empty and singleton values for basic types like integers and vectors [28]. Furthermore, it features functions over data structures, such as ”contains”, ”index_of”, and ”range”, and functions like ”old”, that refer to previous states [28].
3. Choice functionality
MLS implements a ”choice functionality” that lets users obtain values that satisfy user-defined predicates [28].
Thus making MLS a lot more expressive than Move.
The MLS forms the interface to the MVP and provides 2 default types of specifications:
1. global-invariants
Global invariants are contract-level invariants. They are part of a module's definition and are assumed before every state update and have to hold after every state update [28]. Thus Global-invariants are contract-scope. **2. spec blocks**
Spec blocks are blocks of specifications, where specifications refer to assertions (pre and post-state-change) and invariants [28]. There are four types of spec blocks:
a) function spec bocks
These spec blocks are defined separately from the function but have the same name and have access to the pre and post-state of the function [28]. These spec-blocks support preconditions, postconditions and function-invariants, which are translated to pre and post-conditions, thus enabling Hoare-style pre and post-condition verification [28].
b) Struct invariants
These spec blocks, similar to function spec blocks, are defined separately from the struct but share its name [28]. Struct spec-blocks support invariants, preconditions and assertions, which must hold for every function in the struct's module and at the structs initialization [28]. The verification of the Struct's specifications is sound, as Move does allow interaction with the struct from outside the module only through the module's functions [28]. Thus easing the specification and verification of data structures using structs and data captured in structs.
c) inserted spec-blocks
Spec blocks can also be directly inserted in functions [28], where they can be used in two ways:
1. inline assertions
Spec blocks with assertions, inserted in functions, check the assertions at the exact program point, adding to the expressiveness of the MSL [28].
2. loop-invariants
Spec blocks directly above a loop are interpreted as Hoare-style loop invariants that are proven inductively by showing that they hold at the loop entry and that they hold, if assumed before the loop-body, also after the execution of the loop-body [28]. Hence the MVP is unbounded.
d) general spec-blocks General spec blocks can express specifications about the interaction of functions and the behavior of global memory, they are not limited to a single function or struct and are placed freely in the contract, similar to functions [28].

2) Architecture: The Architecture of the MVP is also depicted in Fig. 3, for a quick overview. The MVP takes the Move-contracts, which have the specifications defined in them as input [15]. It then extracts the specifications and translates the Move-sourcecode to Bytecode, on which the verification is performed [15]. The Bytecode and the specifications are translated into a common prover-model on which simplifications and optimizations, such as reference elimination, monomorphization, and more, are performed [15]. The simplified model is then translated into the Boogie IVL, which Boogie further translates into an SMT-Model [7], [22], which is handed to external SMT-solvers, namely CVC5 [9] and Z3 [8] [16]. The SMT-solvers either return the verification or a counter-example, which the MVP maps back to the source code [15], [16], [28].

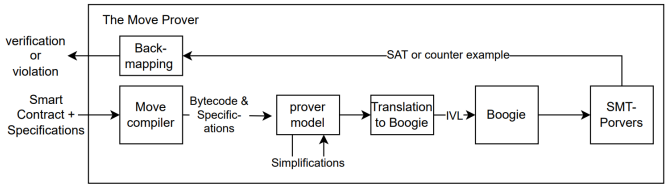

Fig. 3. Move Prover architecture

3) Optimization: The MVP compiles the source code to bytecode, as bytecode-level verification erases the possibility of compiler errors, leading to bugs in formally verified source-code, that was not semantically correctly translated into bytecode, which is what is actually stored and executed on the blockchain [15], [32]. The translation into Boogie's IVL (intermediate verification language) is an optimization as the IVL is optimized for SMT-reasoning but still high-level enough, such that the Bytecode can be translated efficiently into the IVL [15], [22]. Each Opcode is translated into an IVL function (called procedure) [15], [28]. Generic types and functions are represented as Boogie procedures, that take a type as the first parameter [15], [28]. Besides these optimizations, the MVP implements 4 more important simplification and optimization techniques [16]:
1. reference elimination & non-aliasing memory
Move is built on a Rust [29] like type system, which enforces that for every memory location, ”there can be either exactly one mutable reference or n immutable references” [16], [33]. Furthermore, the ”lifetime of references to data on the stack cannot exceed the lifetime of the stack location” [16]. Thus a correctly compiled Move program, which has these properties, can be simplified by reference elimination, meaning every reference can be replaced by the value of the location it's referring to [16]. This complete elimination of references furthermore is a complete non-aliasing memory model [16].
2. monomorphization
Move, in contrast to Solidity, supports generic functions,

which have to be proved for generic types [33]. Monomorphization is a technique that concretizes the generic types into multiple concrete types [16]. Thus replacing the reasoning about abstract generic values with concrete values, which unify to the abstract generic representation [16]. The algorithm to infer the needed types runs on the function and takes into account the types modified and accessed by the function [16]. To start, the MVP skolemizes the generic types into concrete types without any properties, thus representing the abstract generic types in a concrete way, which cuts the need for reasoning about abstract arbitrary types [16]. However, since the types of values an operation is performed on can make a difference in the execution of the operation, based on the types' properties, only generic type instantiations might not be expressive enough [16], [33]. Therefore, specialized types, for these differences in properties, are introduced covering the different possible paths the execution could follow [16]. Once all the specialized types for every operation in the function are computed, the complete type-model of every possible combination of different types for variables and parameters in the function is computed and thus outputs every possible typed path, that needs to be verified to obtain a generic verification [16]. Hence splitting one complex verification goal into multiple easier ones.

3. global invariant injection

The MVP inserts global invariants between individual opcodes, based on static analysis, to simplify the generated verification conditions [16]. The static analysis is based on the memory intersection of functions and the invariant [16]. If the memory accessed by the function does not intersect with the memory accessed by the invariant, the function is not annotated, as it cannot change if the invariant holds or not [16]. If the memory of the function and the invariant do intersect, then the invariant is assumed at the beginning of the function and the invariant is asserted after every opcode that changes memory, that is accessed by the invariant, and after every function call to a function that suspended the invariant, even though it changes the memory the invariant accesses [16]. Note that invariant suspension is an unsound shortcut [33].

4. unsound shortcuts

The MVP implements a few unsound language features, that simply assume some properties - thus they are unsound - but thus also are helping developers in proving their code [28]. Complex proofs can be constructed from simpler proofs, that are based on assumptions, where the assumptions are phased out in the process, thus constructing the complex proof step by step from simpler proofs. Some of these constructs include:

a) Invariant Suspension

Though technically unsound, sometimes invariants hold only before and after functions, not necessarily within them, hence they need to be suspended for a certain amount of steps in the program, after which the invariant holds again [28]. This suspension of sub-parts, after which the invariant holds again, within a function, can result in sound formal verification as the invariant if assumed before the function, holds after the function, thus building a valid Hoare-Triple. However, the feature of suspending an invariant for a complete public function can result in unsound proofs. Note that this suspension is only necessary in some cases and only because of the "Global invariant injection" optimization, which asserts the invariant after every Opcode that changes the memory [28]. A common Hoare-Triple, with an assertion at the end of every function, would not need this shortcut.

b) Abstract Specifications

The MVP allows users to specify arbitrary abstract semantics for functions, that are used by the prover if the function is called by other functions in the proving process [28]. This is always unsound except if the abstract semantics are equivalent to the concrete semantics, which can be defined in the same spec block using "[abstract] and [concrete]" [28].

c) Axioms & uninterpreted functions

The MVP supports "uninterpreted functions", which are functions with an unknown implementation [28]. "Axioms" are a way of constraining these uninterpreted functions, to some specification properties [28]. Which similar to Abstract Specifications might help in reasoning about function calls, specifically external function calls [28]. Though they should be handled with care as they might lead to contradictions, thus corrupting the whole verification (i.e. False $\implies$ everything) [28].

D. Certora Prover

1) Completeness: The Certora prover is a Bytecodelevel formal verification tool, as it performs the verification on the compiled Bytecode of Solidity smart contracts [17]. It takes Solidity smart contracts and a CVL file with the specifications as input [17]. The CVL language is syntactically similar to Solidity and thus provides a corresponding feature for almost all of Solidity's features [19]. Furthermore, it adds a lot of constructs on top to enrich the specification expressiveness, including hooks, ghosts, and quantifiers [19]:

1. Hooks

Hooks are defined on Opcodes and consist of a body of instructions [19]. They are triggered if the Opcode is executed and insert their body of instructions at that point [19]. Thus giving access to low-level EVM reasoning in the specifications.

2. Ghosts

Ghosts are functions that can be used with hooks to extract a program state that is not explicit from the storage variables but can be extracted, thus enriching specification expressiveness [19].

3. Quantifiers

The CVL implements quantifiers in the form of "forall" and "exists" commands [19].

The CVL language is the verification interface for the user and provides two default types of specifications:

1. contract-invariants

Contract invariants are specifications that are invariant in

the contract, meaning they should always hold if nothing is executed [19]. Which Certora realizes by induction: the contract-invariant must hold after the construction and additionally for every function it can be assumed before the execution and must hold after the execution. However, this poses a risk for unsound verification, as Certora's invariants can include external parameters (e.g. time), which can change and thus, invalidate a proven contract-invariant [19].

2. rules

Rules are blocks of commands and post conditions, that can be "filtered", which is prepending a block of preconditions, enabling Hoare-style formal verification [19]. Commands in the rule block can be function calls, but also any other Solidity construct (precisely, CVLs representation according to the Solidity construct) [19]. Post-conditions are realized through asserts that, however, do not revert like Solidity asserts but fail if they are not satisfied [19].

Both types of specifications can or do take into consideration different functions or the whole contract [19] and thus are contract-scope.

Certora is bounded as it uses loop-unrolling instead of loop-invariants. The bound can be set arbitrarily by the user, however, that does not achieve soundness [19].

2) Architecture: The Architecture of the Certora Porver is also depicted in Fig. 4, for a quick overview. The Certora Prover takes two types of inputs, the Solidity files with the source code and the specifications in CVL files [17]. The Source code is first compiled to EVM-Bytecode, which is then compiled to intermediate language TAC [17]. The CVL files are directly compiled into TAC [17]. The model of TAC source code representation and TAC specification representation is then statically analyzed, to produce more easy verification conditions [17]. The result TAC-model of the static analysis is then translated into smt-lib [6] and given to SMT solvers, namely Z3 [8] and CVC5 [9] [17].

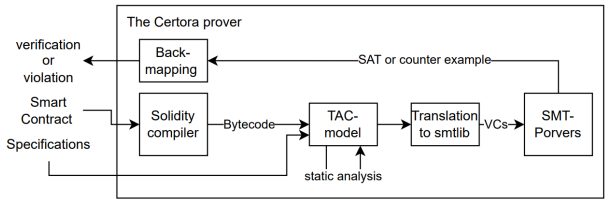

Fig. 4. Certora Porver architecture

3) Optimization: As Certora is a corporate product, the optimization details are mostly broad outlines. However, they include approaches a bit different from the optimizations of most other tools, which might be interesting.

1. intermediate language representation & memory abstraction

The Certora Prover translates the Bytecode into TAC as a memory abstraction [17]. Though no specific steps for non-aliasing properties are taken, the translation to TAC abstracts the stack-based memory model of the EVM into a symbolic value, register-based one, thus easing SMT-solving [17].

2. simplification through static analysis

Certora performs static analysis on the TAC model obtained from the compiled source code and specifications [17], [18]. The Analysis firstly reduces the code to the parts necessary for the verification and secondly infers sound invariants, which help the SMT-solvers as side lemmas, increasing the efficiency of the SMT-solving [17], [18]. The whole analysis is sound, meaning now over or under-approximations are made in the process [17].

3. unsound short-cuts

The Certora Prover provides within the CVL multiple unsound features, that abstract some function or proof away, but also, simplify reasoning for SMT-solvers [19], thus making user interaction much easier, as the user can see under which premises, the proof would go through and thus, what helping lemmas they might need to prove. These unsound features include, among others:

a) Summaries Summaries are declarations of referenced functions, they can be arbitrarily defined and used within the verification and thus are unsound [19]. However, they also pose a gateway to simplify functions to only the parts necessary for the verification, thus creating a tool for mitigating timeouts.

b) axioms for unknown functions The CVL allows "uninterpreted functions" which are functions whose implementation and possibly types are unknown, which allows reasoning about external function calls [19]. Axioms are assumed properties of these functions, but since the functions are unknown, axioms are by construction unsound [19]. However, axioms can help similarly to summaries in easier proof generation and mitigating timeouts.

Besides these optimizations, there is no optimization specifically for reference elimination mentioned.

V. Comparison of Approaches

The tools are compared with regard to Completeness and Optimization, in the sense they were defined in Section 4, meaningful implications are also discussed at the end of each block.

A. Completeness

1) Bytecode/Source code level: All tools expect Sourcecode-level smart contracts as inputs. The Move Prover [15] and the Certora Prover [17] perform their formal verification on the bytecode, while solc-verify [11] and SMT-Checker [14] perform their formal verification on the source code. Therefore, considering the code level the formal verification is performed on, the formal verification of the MVP and Certora Prover seem to be more reliable as it excludes compiler bugs that could still violate the rules proved by solc-verify and SMT-Checker.

2) Contract / Function scope: While solc-verify [11], the MVP [15], and the Certora Prover [17] support contract-wide invariants, which ensure correctness above function level, SMT-Checker [14] does not. However, this does not mean that

TABLE I
APPLICATION

		solc-verify [10], [11], [27]	SMT-Checker [14]	MVP [15], [16]	Certora [17]–[19]
Compleness	unbounded(1)/ bounded(0)	1	0	1	0
	Bytecode(1)/ Sourcecode(0) level	0	0	1	1
	Contract(1)/ Function(0) scope	1	0	1	1
	quantified specifications y(1)/n(0)	1	0	1	1
Optimization	non-aliasing memory (abstraction) y(1)/n(0)	1	0	1	0
	intermediate language repr. y(1)/n(0)	1	0	1	1
	preprocessing wrt. VC simpl. y(1)/n(0)	0	0	1	1
	supported SMT-solvers	Z3 [8], CVC5 [9]	Z3 [8], CVC5 [9]	Z3 [8] ,CVC5 [9]	Z3 [8], CVC5 [9], Yices [20], Vampire [21]

SMT-Checker cannot express such properties. SMT-Checker can be used to ensure contract-wide invariants with a structural induction pattern, by ensuring the invariant holds after the constructor and assumed before every function also holds after every function, similar to Certoras approach [19]. SMT-Checker could therefore be easily upgraded to also support contract-wide invariants and hence become a contract scope tool, however, since it is already replaced by SolCMC [12] this is unlikely and due to tools like solc-verify not being necessary anymore.

3) unbounded / bounded: Solc-verify [11] and the MVP [15] are the only unbounded formal verification tools in this paper [11], [14], [17], [28]. However this does not mean solc-verify and the MVP have novel approaches the others lack. They let the user input loop invariants for Hoare-style reasoning over loops, which is sound and complete, but is no unique or new approach and hinders automaticity [27], [28], [31]. The other two tools, the Certora Porver [19] and the SMT-Checker [14], decided for loop-unrolling, which is replacing the loop with a finite number of repetitions of its body, and unsound [14], [19]. The SMT-Checker does not implement Hoare-reasoning but could be extended to take loop-invariants as specifications (e.g. in comments above the loop) and use inductive SMT theories to prove them. The Certora prover does not support in-code specifications, in favor of portability [19]. The Certora team decided on a specification language in a separate file format which increases the difficulty of annotating a specific loop in a function but also makes specification files easily reusable [18], [19].
Hence formal verification of unbounded loops, as in the Certora prover and the SMT-Checker, should be handled with care. The Certora Prover offers a flag that enables this unsound verification, the ”–optimistic_loop” flag [19]. If this flag is not explicitly set, the verification is still sound, also for unbounded loops.

4) quantified specifications & Specification expressivness: The Move Prover [15] and the Certora Prover [17] are the provers with the greatest specification richness, as they both allow quantified contract-wide invariants as well as additional advanced language-constructs in their specification languages [19], [28].
All tools support Hoare-style pre and post-conditions [14], [19], [27], [28]. Solc-verify, the MVP and the Certora prover furthermore support contract-wide invariants, quantifiers (i.e. forall and exists expressions) and a simple extension over the source code language in the form of functions over data structures, such as ”sum” and ”range”, and default values, such as ”MAX” and ”MIN” [14], [19], [27], [28].
Further advanced Language features are only supported in the MVP and the Certora Prover [19], [28]. Both, the Certora Prover and the MVP support unsound-short cuts in their provers and thus their specification languages [19], [28]. Unsound-short cuts are unsound language constructs that assume some properties for the provers, thus easing the verification process. They can be helpful for developers in creating complex proofs. Function abbreviations, a type of unsound shortcuts, replace a function’s implementation with an arbitrary specification. The Certora Prover and the MVP, both implement function abbreviations, called ”Summaries” in Certora [19] and ”Abstract functions” in the MVP [28]. Axioms, another type of unsound shortcut, are properties assumed for functions with an unknown implementation. The MVP and the Certora Prover also both implement Axioms, under the same name [19], [28]. The MVP also supports invariant suspension as an unsound shortcut [28]. Suspending an invariant means not enforcing the invariant for a section of commands or a function, which is sometimes needed even for sound formal verification in the MVP, however, nothing the Certora Prover lacks, as its architecture is simply built different, such that invariant suspension is not needed [18], [19], [28].
Besides unsound shortcuts, the Certora Prover offers Hooks and Ghosts, which allow inspection and interaction with the Bytecode [19]. While the MVP does not offer such low-level interaction, it implements a Choice-functionality, which allows obtaining a value with properties given in the specification language, which is a lot richer than source code (e.g. offering the possibility to obtain a function f that satisfies a property P like ”forall x. P (f x)”) [28]. The Certora Prover does not implement a Choice-functionality [19].

B. Optimization

1) non-aliasing memory abstraction: Memory abstraction is a widely used optimization technique, as 3 out of the 4 tools compared in this paper use some kind of it, namely solc-verify, the MVP, and the Certora Prover [10], [16], [17]. While the Certora-Prover replaces stack-based references with symbolic value registers, to simplify reasoning [17], solc-verify and the MVP go a step further and try to eliminate alias reasoning for references, where possible [10], [16]. This elimination is made possible by their non-aliasing memory modulation. The MVP achieves a complete non-aliasing memory model for every reference, as the underlying language Move, supports a Rust-like type system with a borrow-checker and thus ownership of memory for references [16], [33]. Solc-verify on the other hand works on Solidity-sourcecode, which does allow multiple references that can change the same memory space [26] therefore, making a complete non-aliasing memory model for every reference impossible. However parts of the Solidity-language support non-aliasing, namely the storage part of Solidity's memory model [10], [26]. Solc verify uses the non-aliasing properties of the storage-memory and extends it to also cover storage pointers where ever possible [10].

2) preprocessing wrt. VC simplification: All tools simplify the source code to some point, be it compiling to Bytecode or using the AST from the compiler and thus utilizing compiler simplifications to generate simpler VCs [11], [14], [15], [17]. However, simplification beyond the compiler scope, before or after the translation to the SMT model, is only performed by the MVP [16] and Certora [17].
Certora uses Static Analysis on their TAC representation of the Bytecode to infer "non-trivial invariants" [17]. Thus adding to the set of assumptions the SMT-solvers can use for reasoning and eliminating the proof for simple invariants. Hence simplifying the VCs and thus the proving for SMT-solvers.
While Certora uses static analysis to add assumptions, the MVP implements a simplification that simplifies the verification target, the code and the specifications [16]. The MVP replaces all references with concrete values, made possible by its complete non-aliasing memory model [16]. Thus eliminating the need for SMT-solvers to reason about pointers and simplifying the VCs, as they are solely dependent on values, that are not bound to any location or variable. The MVP also simplifies the verification of generic function by splitting it into multiple verification goals with different type instantiations [16], which is not necessary for Solidity as it does not support generic functions [26].

3) intermediate language representation: Intermediate language representation is a common optimization, as 3 out of the 4 tools compared in this paper make use of it. SMT-Checker is the only tool, that translates source code directly to the format the SMT-APIs expect [14].
The MVP [15] and solc-verify [11] translate their code to the Boogie IVL, a language that is designed to be syntactic close to programming languages but is also optimized for SMT-solvers [7], [11], [16], [22]. Hence the MVP and solc-verify use the Boogie IVL as an SMT-efficiency optimization.
The Certora Prover translates the stack-based Bytecode into TAC, a register-based language [17]. Hence Certora uses the intermediate language TAC as a memory abstraction.

VI. Discussion

The state-of-the-art did evolve rapidly since the first publication in 2018 (SMTChecker), which worked only with Solidity's "assert" and "requires" for pre and post-conditions [14], to tools like the Move Prover (MVP) (2022), which supports quantifiers, loop invariants, and program invariants and still is almost as fast as testing [15], [16]. Optimizations like a non-aliasing memory representation and translation into an intermediate language, which is easier to translate to SMT-solvers, have been implemented [10], [16], [17].
Despite the advancement, three are still open work areas. There is only one SMT-based tool per language that supports the current version (the Certora Prover [17], which is not open-source, for Solidity and the Move Prover [28] for Move). Furthermore, the Certora Prover does not support loop invariants [19], which makes it impractical for real-world formal verification and effectively more of a bug-hunting tool, than a formal verification tool. Thus leaving Solidity with no SMT-based provers, even though it is the most popular smart contract language [24].
The Move Porver remains, hence, as the only practical SMT-based tool for formal verification, representing the state-of-the-art.
It has to be noted that the approach of this paper is limited in that it does not take alternative formal verification approaches, like CHCs or interactive theorem provers, into consideration, which, however, also allowed to cover SMT-based optimizations in much greater depth.
There are no direct performance comparisons in this paper, which would have been nice, however, since the tools support different versions of languages (Certora for Solidity 0.8 [18], solc-verify for Solidity 0.6 and 0.7 [27]) and even different languages (Move Porver for Move [15]) defining a common test set was impossible. Furthermore, the result would have been probably obvious, as the Certora Prover and the MVP are the most advanced tools, they would probably be the fastest, with the MVP a bit faster, as Move was designed for formal verification [15] with the MVP, whereas Solidity was not designed for formal verification at all [26]. Additionally, there is already a performance comparison of SMT-Checker and solc-verify, where solc-verify is faster [11].
I hope to give through this paper a broad understanding of the quite new field of SMT-based formal verification of smart contracts and provide a starting point for further research and development.

VII. Related work

The closes related work to the tools in this paper are static-analysis SMT-based tools, like Oyente [42], VeriSol [45],

MAIAN [44], and GASPER [43]. However, they all have an over-approximating or domain-specific approach and thus are not suitable for formal verification [42]–[45].

There are multiple SoK papers, that investigated formal verification on smart contracts, however, none are specific about SMT-based methods and all are written in a more high-level approach. [37] discusses which formal methods, in general, are applicable for smart contracts and does not mention SMT at all. [38] discusses formal verification of smart contracts over a broader scope, including SMT/SAT based-approaches, which have a category in the paper. The tools mentioned there, however, are mostly the static-analysis tools that are not suitable for formal verification. Furthermore, the paper only focuses on open-source Solidity-language tools, leaving out important SMT-based approaches like the Certora prover and the MVP, which are covered in this paper. [39] discuss only static analysis based on SMT-solvers.

Another category related to the work in this paper are Constrained Horn Clause (CHC) based formal verification approaches, which the successors of SMTCHecker SolCMC [12] and Solicitous [13] use. CHCs are an alternative to Hoare-logic but need CHC-specific solvers [12], which is why a comparison with regard to optimization would not have fitted into this paper. CHC solvers, due to their specific task (solving only CHCs), are also not as developed as SMT-solver, though some SMT-solver like Z3, start supporting CHCs [8].

VIII. Conclusion

This paper presented the first overview of SMT-based formal verification tools for smart contracts.

The most recent tools, the Certora Prover and the Move Prover, implement separate specification languages, which enrich source-code expressions among others with quantifiers, uninterpreted functions, and more, dependent on the prover. They support Hoare-style pre and post-conditions and program invariants, partly they also support loop invariants.

The most common and impactful optimizations are non-aliasing memory representation, an abstraction of the code into an SMT-friendlier intermediate language and simplifications before the actual SMT-solving in the form of replacing references with values and using static analysis to infer invariants that can be used as assumptions, from the code.

Furthermore, it was found, that per language there exists only one SMT-based formal verification tool and that the one for Solidity, the Certora Prover, which is a corporate product, does not even support loop-invariants. Thus presenting an opportunity for future work. SMT-based solvers present an efficient way to perform formal verification and the MVP could serve as an example for a similar complete SMT-based prover for Solidity. Though one of the MVP's most impactful optimizations, the replacing of references with values, relies on the non-aliasing memory properties of Move, the extensive partly non-aliasing memory formalization of solc-verify could be reused and utilized to achieve a similar optimization while supporting loop-invariants. Though the approach needs further investigations, it could redefine security in the Ethereum ecosystem, as formal verification could become part of a developer's process in programming smart contracts. Thus significantly increasing the trust and security in the space.

References

[1] V. Buterin. "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. By Vitalik Buterin (2014)". Ethereum.org. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf (accessed Jun. 5th, 2023)

[2] T. Nipkow, G. Klein, "Hoare-Logic" in Concrete Semantics with Isabelle/HOL, 1st ed. Munich, Germany: Springer Verlag, 2014, ch. 12, sec. 12.2, pp. 191–193. Accessed: Jun 5th, 2023. [Online]. Available: http://www.concrete-semantics.org/.

[3] H. Barbosa, A. Reynolds, D. El Ouraoui, C. Tinelli, C. Barrett, "Extending SMT Solvers to Higher-Order Logic" in Fontaine, P. (eds) Automated Deduction – CADE 27, 1st ed. Switzerland, Springer, Cham, 2019 pp. 35-54. Accessed: Jun 5th, 2023. [Online]. Available: https://doi.org/10.1007/978-3-030-29436-6_3.

[4] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia and Cesare Tinelli, "Satisfiability Modulo Theories" in Handbook of Satisfiability, Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsch, 2nd Ed., Amsterdam, Netherlands: IOS Press, 2009, ch. 12, pp. 825-873

[5] Charalambos Mitropoulos, Thodoris Sotiropoulos, Sotiris Ioannidis and Dimitris Mitropoulos, "Syntax-Aware Mutation for Testing the Solidity Compiler" unpublished. Accessed: Jun 9th, 2023. [Online]. Available: https://dimitro.gr/assets/papers/MSIM23.pdf.

[6] "SMT-LIB The Satisfiability Modulo Theories Library." http://smtlib.cs.uiowa.edu/ (accessed Jun 11, 2023).

[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs', in Formal Methods for Components and Objects, 2006, pp. 364–387.

[8] L. de Moura and N. Bjørner, 'Z3: An Efficient SMT Solver', in Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 337–340.

[9] H. Barbosa et al., 'cvc5: A Versatile and Industrial-Strength SMT Solver', in Tools and Algorithms for the Construction and Analysis of Systems, 2022, pp. 415–442.

[10] Á. Hajdu and D. Jovanović, 'SMT-Friendly Formalization of the Solidity Memory Model', in Programming Languages and Systems, 2020, pp. 224–250.

[11] Á. Hajdu and D. Jovanović, 'solc-verify: A Modular Verifier for Solidity Smart Contracts', in Verified Software. Theories, Tools, and Experiments, 2020, pp. 161–179.

[12] L. Alt, M. Blicha, A. E. J. Hyvärinen, and N. Sharygina, 'SolCMC: Solidity Compiler's Model Checker', in Computer Aided Verification, 2022, pp. 325–338.

[13] R. Otoni, M. Marescotti, L. Alt, P. Eugster, A. Hyvärinen, and N. Sharygina, 'A Solicitous Approach to Smart Contract Verification', ACM Trans. Priv. Secur., vol. 26, no. 2, Mar. 2023.

[14] L. Alt and C. Reitwiessner, 'SMT-Based Verification of Solidity Smart Contracts', in Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice, 2018, pp. 376–388.

[15] J. E. Zhong et al., 'The Move Prover', in Computer Aided Verification, 2020, pp. 137–150.

[16] D. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu, and E. Zhong, 'Fast and Reliable Formal Verification of Smart Contracts with the Move Prover', arXiv [cs.PL]. 2022.

[17] Certora Whitepaper, (2022). Accessed: Jun 12, 2023. [Online]. Available: https://docs.certora.com/en/latest/docs/whitepaper/

[18] The Certora Prover, (2022). Accessed: Jun 12, 2023. [Online]. Available: https://docs.certora.com/en/latest/docs/prover/

[19] The Certora Verification Language, (2022). Accessed: Jun 12, 2023. [Online].https://docs.certora.com/en/latest/docs/cvl/

[20] B. Dutertre, 'Yices 2.2', in Computer Aided Verification, 2014, pp. 737–744.

[21] Vampire.(v4). Accessed: Jun 12, 2023. [Online]. Available: https://vprover.github.io/

[22] K. R. M. Leino and P. Rümmer, 'A Polymorphic Intermediate Verification Language: Design and Logical Encoding', in Tools and Algorithms for the Construction and Analysis of Systems, 2010, pp. 312–327.

[23] D. Monniaux, 'Quantifier Elimination by Lazy Model Enumeration', in Computer Aided Verification, 2010, pp. 585–599.

[24] “2023 Developer Survey”. Stackoverflow.com. https://survey.stackoverflow.co/2023/#most-popular-technologies-language (accessed Jun 14, 2023).
[25] Solidity. Accessed: Jun 14, 2023.[Online]. Available: https://soliditylang.org/
[26] Solidity.(v0.8.20). Accessed: Jun 14, 2023.[Online]. Available: https://docs.soliditylang.org/en/v0.8.20/
[27] solc-verify. (2020). [Online]. Accessed: Jun 16, 2023. Available: https://github.com/SRI-CSL/solidity/
[28] Move. [Online]. Accessed: Jul 12, 2023. Available: https://github.com/move-language/move
[29] Rust. [Online]. Accessed: Jul 12, 2023. Available: https://www.rust-lang.org/
[30] CVC4. [Online]. Accessed: Jul 12, 2023. Available: https://cvc4.github.io/
[31] R. W. Floyd, ‘Assigning Meanings to Programs’, in Program Verification: Fundamental Issues in Computer Science, T. R. Colburn, J. H. Fetzer, and T. L. Rankin, Eds. Dordrecht: Springer Netherlands, 1993, pp. 65–81.
[32] Aptos-whitepaper. [Online]. Accessed: Jul 14, 2023. Available: https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf
[33] The Move Book. [Online]. Accessed: Jul 12, 2023. Available: https://move-book.com/
[34] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang, ”Symbolic model checking: 10/sup 20/ states and beyond,” [1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, PA, USA, 1990, pp. 428-439, doi: 10.1109/LICS.1990.113767.
[35] C. Barrett and C. Tinelli, ‘Satisfiability Modulo Theories’, in Handbook of Model Checking, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Cham: Springer International Publishing, 2018, pp. 305–343.
[36] H. Chen, M. Pendleton, L. Njilla, and S. Xu, ‘A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses’, CoRR, vol. abs/1908.04507, 2019.
[37] M. Krichen, M. Lahami and Q. A. Al–Haija, ”Formal Methods for the Verification of Smart Contracts: A Review,” 2022 15th International Conference on Security of Information and Networks (SIN), Sousse, Tunisia, 2022, pp. 01-08, doi: 10.1109/SIN56466.2022.9970534.
[38] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, ‘A Survey on Formal Verification for Solidity Smart Contracts’, in Proceedings of the 2021 Australasian Computer Science Week Multiconference, Dunedin, New Zealand, 2021.
[39] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, ‘On the Verification of Smart Contracts: A Systematic Review’, in Blockchain – ICBC 2020, 2020, pp. 94–107.
[40] defillama, Jul 16, 2023. [Online]. Available: https://defillama.com/
[41] Consensys best practices, Jul 16, 2023. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/
[42] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, ‘Making Smart Contracts Smarter’, in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 2016, pp. 254–269.
[43] T. Chen, X. Li, X. Luo and X. Zhang, ”Under-optimized smart contracts devour your money,” 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 2017, pp. 442-446, doi: 10.1109/SANER.2017.7884650.
[44] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, ‘Finding The Greedy, Prodigal, and Suicidal Contracts at Scale’, CoRR, vol. abs/1802.06038, 2018.
[45] S. K. Lahiri, S. Chen, Y. Wang, and I. Dillig, ‘Formal Specification and Verification of Smart Contracts for Azure Blockchain’, CoRR, vol. abs/1812.08829, 2018.