The Treewidth of Smart Contracts

Krishnendu Chatterjee
IST Austria
Klosterneuburg, Austria
krishnendu.chatterjee@ist.ac.at

Amir Kafshdar Goharshady
IST Austria
Klosterneuburg, Austria
amir.goharshady@ist.ac.at

Ehsan Kafshdar Goharshady
Ferdowsi University of Mashhad
Mashhad, Iran
e.kafshdargoharshady@mail.um.ac.ir

ABSTRACT

Smart contracts are programs that are stored and executed on the Blockchain and can receive, manage and transfer money in the form of cryptocurrency units. Two important problems regarding smart contracts are formal analysis and compiler optimization. Formal analysis is extremely important, because smart contracts hold funds worth billions of dollars and their code is immutable after deployment. Hence, an undetected bug can potentially cause significant financial losses. Compiler optimization is also crucial, because every action of a smart contract has to be executed and verified by every node in the Blockchain network. Hence, optimizations in compiling smart contracts can lead to significant savings of computation, time and energy.

Two classical approaches in both program analysis and compiler optimization are *intraprocedural* and *interprocedural* analysis. In intraprocedural analysis, each function is analyzed separately, while interprocedural analysis considers the entire program. In both cases, optimization and analysis problems are often reduced to graph problems over the control flow graph (CFG) of the program. However, the resulting graph problems are often computationally expensive. Hence, there has been ample research on exploiting structural properties of CFGs to obtain efficient algorithms for these problems. One well-studied structural property is the treewidth. Treewidth is a measure of tree-likeness of graphs and small treewidth can be exploited for efficient algorithms. It is known that intraprocedural CFGs of structured programs have treewidth at most 6, whereas the interprocedural treewidth cannot be bounded. Bounded treewidth has been used as a basis for many efficient intraprocedural analyses.

In this paper, we explore the idea of exploiting the treewidth of smart contracts for formal analysis and compiler optimization. First, similar to classical programs, we show that the intraprocedural treewidth of structured Solidity and Vyper smart contracts is at most 9. Second, for global analysis, we prove that the interprocedural treewidth of structured smart contracts is bounded by 10 and, in sharp contrast with classical programs, treewidth-based algorithms can be easily applied for interprocedural analysis. Finally, we supplement our theoretical results with experiments using a tool we implemented for computing treewidth of smart contracts and show that the treewidth is much lower in practice. We use 36,764 real-world Ethereum smart contracts as benchmarks and find that they have an average treewidth of at most 3.35 for the intraprocedural case and 3.65 for the interprocedural case.

CCS CONCEPTS

• **Applied computing** → **Secure online transactions**; • **Software and its engineering** → **Formal software verification**;

KEYWORDS

Smart Contracts, Parameterized Algorithms, Treewidth, Blockchain, Compiler Optimization, Program Analysis, Control Flow Graphs

1 INTRODUCTION

In this paper, we study the possibility of exploiting the treewidth property of (global) control flow graphs of Ethereum smart contracts for solving both intraprocedural and interprocedural static analysis problems. We first obtain sharp theoretical constant bounds on the treewidths, and then provide an extensive experimental evaluation, showing that, in practice, the treewidth is always very small.

Blockchain and Bitcoin. Blockchain was first developed as a tamper-proof decentralized ledger for enforcing consensus about transactions in Bitcoin [39]. However, it was soon realized that Blockchain (and its extensions) can have much wider use cases and can enforce any kind of well-defined consensus [44]. This led to hundreds of new Blockchains and cryptocurrencies, each with their own unique attributes and advantages over the classical Bitcoin Blockchain. However, Bitcoin remains the largest cryptocurrency, with a market cap of just over 100 billion dollars [19].

Transaction Scripts. Bitcoin transactions can include simple scripts that set the necessary conditions for a party to be able to claim and use the funds in the transaction [5]. In the most basic case, a transaction usually includes the public key of its recipient and the transaction script asks for a valid signature corresponding to that public key to allow access to the funds. However, one can set more complicated conditions. For example, a transaction script might ask for signatures from at least two of three predefined parties. A notable use-case of these scripts is to enforce contract-like behavior. For example, BitHalo is a protocol based on Bitcoin scripts that can replace intermediaries and provide escrow services [51].

Smart Contracts. The idea of encoding semantics in cryptocurrency transactions can be extended to more complicated programs than Bitcoin scripts, in order to handle more complex financial agreements, such as credit reporting [30] or decentralized autonomous organizations [18]. In general, a smart contract is a program that is executed on the Blockchain. A contract has its own dedicated memory and can be programmed to receive, manage and transfer cryptocurrency units [38]. Users (and other contracts) can interact with the contract by calling one of its functions. Each such function call is handled in the same way as a cryptocurrency transaction. The Blockchain protocol provides a global consensus about the state of each contract. This includes consensus about the code of the contract, its semantics, the state of the contract's memory, and the results of interactions with the contract. Note that after a contract's code is stored on the Blockchain, it is immutable, and the only way to interact with it is to call its functions, which behave as programmed at the time of deployment.

Ethereum. Ethereum is a cryptocurrency platform that allows arbitrary Turing-complete smart contracts [9]. This is achieved by means of a well-defined Virtual Machine and an Assembly-like Bytecode format [49]. There are many languages for writing Ethereum contracts. Contracts programmed in these languages are compiled to Ethereum Bytecode before deployment on the Blockchain. Ethereum has become the second-largest cryptocurrency platform in terms of market cap (more than 35 billion dollars at the time of writing), and is the most popular platform for smart contracts [19].

Example. Consider the contract shown in Figure 1. This contract is written in Solidity, which is the most widely-used language in Ethereum. The creator of the contract (the programmer), first puts it on the Blockchain. The creation of this contract is recorded on the Blockchain in the same manner as a transaction and is subject to consensus. When the contract is added to the Blockchain, anyone on the network can see it and interact with it by calling its functions. Each function call is also treated as a transaction and its parameters and the resulting changes to the smart contract are recorded on the Blockchain. Note that in order to achieve a consensus, every node of the Blockchain network has to execute the function (unless the function only reads contract data, and does not make any changes).

The contract has two functions, `deposit` and `submitSolution`. The contract also holds two variables in its dedicated memory, namely `balance` and `N`. Anyone on the network can call the function `deposit`, providing a value for the parameter `_N`. The keyword `payable` signifies that one can pay an arbitrary amount of cryptocurrency units to the contract at the time of calling this function. The amount of received cryptocurrency units can be accessed using `msg.value`. A call to this function sets the value of the variable `N` and increases the `balance` of the contract, i.e. the amount of cryptocurrency units under the contract's control. These cryptocurrency units can only be redeemed if the contract code allows it. The contract rewards anyone who can factor the given number `N`. Anyone on the network can call the `submitSolution` function and provide a candidate factorization. The contract checks the factorization and rewards the caller iff it is correct. Note that the person setting the reward, as well as everyone else, can see all the transactions (function calls) on the Blockchain and can hence obtain the factorization.

Value of Contracts. Smart Contracts hold and manage a considerable amount of funds. At the time of writing, in Ethereum alone, there are over a million instances of deployed smart contracts, holding billions of dollars of funds [46]. Specifically, there is a single contract on the Ethereum Blockchain, that currently holds more than 300 million dollars [46].

Importance of Formal Analysis. Given the unmalleability of data stored on the Blockchain, a contract's code cannot be amended after its deployment. Similarly, all transactions stored in a Blockchain are irreversible. On the other hand, contracts handle a huge amount of funds. Therefore, to avoid significant financial losses, bugs in smart contracts must be detected before deployment [4, 12, 15, 40]. For example, in one catastrophic case, called the DAO attack [18], an attacker stole more than 50 million dollars from a contract. There are also a variety of proposed best-practices and design patterns for minimizing the damages when a contract fails (e.g. see [20]).

```
contract factor {

uint balance;
uint N = -1;

function deposit(uint _N) payable {
  if(N == -1) { //Do nothing if N is already set
    N = _N; //Set the value of N
    balance += msg.value; //Update the balance
  }
}

function submitSolution(uint p, uint q) {
  if(p>1 && q>1 && p*q==N) { //Check correctness
    msg.sender.send(balance); //Pay reward
    balance = 0; //Update the balance
  }
}
}
```

Figure 1: A Smart Contract that rewards factoring a number.

Importance of Optimization. To achieve consensus, each time a function is called, all nodes of the Ethereum network have to verify the results by running the function [49]. This means that each function call is executed and verified by tens of thousands of nodes in parallel [28]. Therefore, any optimization in the compilation process can lead to a massive overall saving of time and energy.

Importance of Quantitative Analysis. In many scenarios, qualitative analysis is not enough for smart contracts [15]. For example, absolute safety against any attacks might be impossible or very costly. In such scenarios, one would like to find a bound on the potential or expected economic consequences of an attack. Moreover, quantitative analysis approaches, such as [17], can be used to find the expected execution costs of a contract.

Control Flow Graphs. Many problems in program analysis, model checking and compiler optimization can be reformulated as graph problems [37]. In such cases, the underlying graph is usually the Control Flow Graph (CFG) of the program that is being analyzed or optimized [35, 37]. See Section 2.1 for a formal definition of CFGs. In such analyses, one can either consider the problem over individual functions and hence create a separate CFG for each function, or attempt to solve the problem on a global control flow graph (GCFG) of the entire program. The former approach is called *intraprocedural* analysis, and the latter is *interprocedural* analysis.

Exploiting Structural Properties. The graph problems arising from formal program analysis and compiler optimization are often computationally expensive and even NP-hard in many cases. Hence, there has been ample research on exploiting the structural properties of the underlying CFGs to obtain faster algorithms [3]. An extensively-studied parameter that has been applied successfully to these problems is the treewidth [8, 13, 16, 32, 36, 41, 47]. See section 2.3 for some motivating examples.

Treewidth. Treewidth [42] is a well-studied graph parameter that provides a measure of tree-likeness of graphs [7]. Trees and forests are the only graphs with a treewidth of 1 and, informally, a lower treewidth means that the graph has more resemblance to trees. The significance of treewidth in the design of algorithms stems from the fact that many NP-hard graph problems are fixed-parameter tractable when parameterized by the treewidth, i.e. can be solved

efficiently on graphs that have small treewidth [6, 7, 11, 23, 24, 29, 31]. Specifically, one can apply a bottom-up dynamic programming technique on such graphs in a manner very similar to trees [6]. For the formal definition of treewidth, see Section 2.4.

Treewidth of Control Flow Graphs. In [47], it was established that the intraprocedural CFGs of goto-free structured programs in several languages, including C and Pascal, have a treewidth of at most 6. This result provided a basis for the fast formal analysis and compiler optimization algorithms mentioned earlier. While general Java programs do not have constant treewidth, in [32], it was shown that real-world Java programs have bounded treewidth and that, in practice, they lead to an average treewidth of less than 3.

Treewidth of Interprocedural Control Flow Graphs. While bounded treewidth has been successfully exploited to obtain faster analysis and optimization algorithms, the boundedness results only hold for the treewidth of intraprocedural CFGs, i.e. CFGs modeling a single function or procedure. It is known that global (interprocedural) treewidth boundedness can lead to many computational advantages [29], but different representations of global CFGs of most real-world programs are either infinite or do not have bounded treewidth and hence, exploiting global treewidth boundedness is not considered to be a realistic approach [29].

Our Contribution. We study the treewidth of structured, i.e. goto-free, Ethereum smart contracts. We focus on two contract programming languages, namely Solidity and Vyper. Solidity is currently the most widely-used language for writing smart contracts. Vyper is the newest language developed by Ethereum foundation and is expected to be widely adopted in near future. We obtain the following results:

(i) Theoretical Results. First, for intraprocedural analysis, we show that CFGs of smart contracts have a treewidth of at most 9 (Theorem 3.1). This is similar to the result for classical programs. Second, in contrast to the results for classical programs, for global (interprocedural) analysis, we show that the global CFGs of Vyper smart contracts are finite, have a treewidth of at most 10 (Theorem 4.1), and their tree decompositions can be succinctly represented (Theorem 4.2). The same result also holds for non-recursive Solidity smart contracts. Hence, unlike classical programs, for smart contracts, solving global variants of many important formal analysis and compiler optimization problems is no harder than the local (intraprocedural) variants.

(ii) Experimental Results. On the experimental side, we implemented a tool for obtaining tree decompositions of CFGs and GCFGs of Solidity smart contracts. We analyzed 36,764 real-world Solidity smart contracts currently deployed on the Ethereum Blockchain. The results showed that (i) no real-world Solidity smart contract used recursion, hence all of our theoretical results for Vyper contracts carry on to real-world Solidity contracts, (ii) in case of CFGs, the average treewidth was 3.35 and the bound 9 was never met, (iii) in case of GCFGs, all analyzed contracts were shown to have finite and succinctly-representable GCFGs. The average GCFG treewidth was 3.65 and the bound 10 was never met in practice.

Significance of Our Results. There are two important takeaways from our results. First, much like classical programs, the bounded treewidth property of the CFGs of smart contracts can be exploited for intraprocedural compiler optimization and program analysis tasks. Second, unlike classical programs, smart contracts have succinctly representable GCFGs with small treewidth. This means that treewidth can be exploited for the same optimization and formal analysis problems in an *interprocedural* setting, rather than just intraprocedural analysis. This can potentially lead to much more powerful algorithms and tools for the analysis of smart contracts. Note that, while exploiting treewidth in general programs is well-studied and supported by tools such as [14], current approaches for analyzing smart contracts, such as [15, 40, 48], do not exploit the treewidth or any other structural property.

Organization. We provide our definitions, formalize our notation and review some previously-known results in Section 2. Our theoretical results on CFGs are presented in Section 3, followed by our theoretical results on GCFGs in Section 4. Finally, we report on our tool and experimental results in Section 5.

2 PRELIMINARIES AND MODELING OF PROGRAMS

In this section, we provide some basic definitions, define an abstract programming language that we are going to use in the rest of the paper, and review previously-known results.

2.1 Programs and Graphs

Control Flow Graphs (CFGs). The Control Flow Graph (CFG) of a program is a directed graph whose paths model the execution traces of the program [1]. There are many slightly different variations of CFGs. The nodes of a CFG can correspond to statements in the program, or basic blocks, or other subdivisions of the code. We will follow the node structure used in [47] (explained below).

Abstract Programming Language. We define an extension of the STRUCTURED programming language [47] to capture the general properties of goto-free programs, abstract away the details that are not relevant to the treewidth of the CFG, and obtain general results that will then be instantiated for specific programming languages. We call this extension ES (Extended STRUCTURED). The differences between ES and STRUCTURED are that (i) to enable interprocedural analysis, ES does not abstract away function calls, and (ii) ES allows several exit types for every function. Informally, each function in ES is like a program in STRUCTURED.

ES. A program in ES is a set of functions. Each function starts with the keyword `function`, followed by its name, followed by a sequence of statements in the function, and finally ends with the keyword `endfunction`. The statements are of these types:

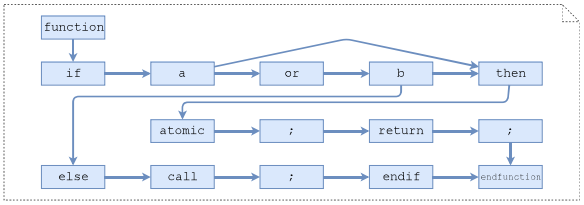
- Conditional statements, `if-then-endif` and `if-then-else-endif`,
- A general loop structure `loop-endloop` that does not normally end on its own,
- A `break` statement, ending the innermost surrounding loop,
- A `continue` statement that goes to the next iteration of the innermost surrounding loop,
- A `return` statement that terminates the current function and returns control to the parent function,

- Several *types* of `exit` statements, $\texttt{exit}_1, \texttt{exit}_2, \ldots, \texttt{exit}_k$, each of which terminates the program,
- Function call statements `call <function-name>`, and
- Atomic statements shown by the keyword `atomic`. By an `atomic`, we mean a statement that does not affect the flow of the program and can be abstracted away, e.g. an assignment.

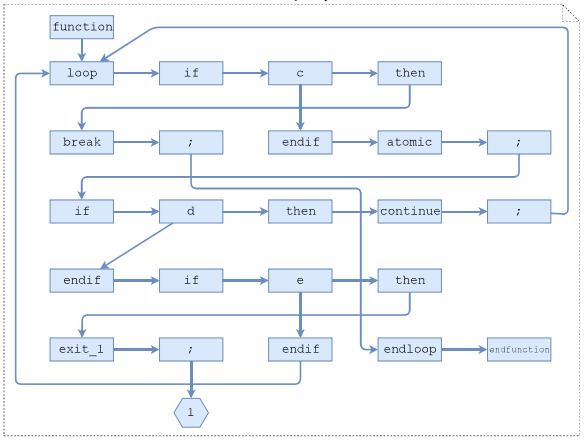
All statements that lack an "end" in their structure must be followed by a semicolon. Intuitively, the different types of `exit` statements correspond to the different types of termination that can happen in a smart contract (see Section 2.2). We assume that a `break` or `continue` that is not surrounded by a loop acts as a `return`. As in STRUCTURED, the conditions of `if` statements can be boolean expressions consisting of atomic boolean variables `a`, `b`, ..., and the operators `and` and `or`. We also assume short-circuit evaluation.

```
function f1
  if a or b then
    atomic;
    return;
  else
    call f2;
  endif
endfunction
```

```
function f2
  loop
    if c then
      break;
    endif
    atomic;
    if d then
      continue;
    endif
    if e then
      exit_1;
    endif
  endloop
endfunction
```



CFG(f1)



CFG(f2)

Figure 2: An ES program (top) consisting of two functions `f1`, `f2` and the CFG of `f1` (middle) and that of `f2` (bottom). Exit nodes are shown using hexagons and nodes that correspond to words in the ES program are shown by rectangles.

Nodes of a CFG. As in [47], we construct the control flow graph for each function separately. In the CFG, we put one vertex for each exit type and one for every word of the ES code, except for function names[1]. Note that this includes semicolons. In the CFG, there is an edge between two vertices, if their corresponding words are in the same function and can be visited consecutively in some execution of the ES program, i.e. in CFGs we treat `call` statements in the same manner as `atomic` statements[2]. Figure 2 shows an example ES program together with its CFG. The nodes corresponding to exit types are shown by hexagons. We will refer to them as hexagonal nodes in the sequel.

Call Graphs and Recursive Functions. The call graph [43, 45] of a program is a directed graph, in which there is one vertex corresponding to each function, and there is a directed edge from a vertex u to a vertex v, if the function corresponding to u, at some point, calls the function corresponding to v. For example, the call graph of the program in Figure 2 has a single edge `f1` $\rightarrow$ `f2`. If the call graph contains a cycle, we say that the program is *recursive* [26]. Otherwise, the call graph is acyclic, and the program is said to be nonrecursive or *simple*.

CFGs are usually sufficient for intraprocedural analysis. However, in order to perform interprocedural analysis the data encoded by the call graph becomes necessary, too. This complicates the situation, given that the analysis problems are now reduced to graph problems over two graphs with nontrivial interactions. We can mitigate this problem by using a Global (interprocedural) CFG. Intuitively, the process for obtaining the GCFG from the CFG and the call graph is very similar to repeated inlining [10].

Global Control-Flow Graphs (GCFGs). The GCFG of a program is obtained from its CFG by repeatedly expanding the function `call` nodes with copies of the CFG of the function that is being called at that point. This process is continued as long as there are unexpanded function `call` nodes remaining in the graph. Concretely, in case of ES programs, a function call `call f;` is expanded by putting a copy of the CFG of `f` between the `call` node and its corresponding semicolon. Specifically, the edge between the `call` node and its semicolon node is removed, the `call` node is connected to the `function` node of the copy and the `endfunction` node is connected to the semicolon. This is illustrated in Figure 3. Intuitively, each node of the GCFG encodes not only a point of the code, but also the functions that are on the stack when reaching that point. So, the GCFG of a program is finite iff the program is simple.

2.2 Ethereum

Ethereum is a cryptocurrency platform that allows smart contracts of arbitrary, i.e. Turing-complete, complexity. Smart contracts are fundamental to Ethereum, to the extent that Ethereum creators often call it a distributed computing platform [9], considering smart contracts as "decentralized applications" and the currency, Ether, as a mechanism of payment in exchange for consensus and computation. Therefore, it is of paramount importance that every node in the Ethereum network has the exact same understanding about the

[1]This definition of nodes is very fine-grained. However, Lemma 2.2 shows that one can contract any two vertices to get a coarser CFG, without increasing the treewidth.
[2]This is because, as in [47], the CFGs are meant to be used for intraprocedural analysis. We use GCFGs for interprocedural tasks.

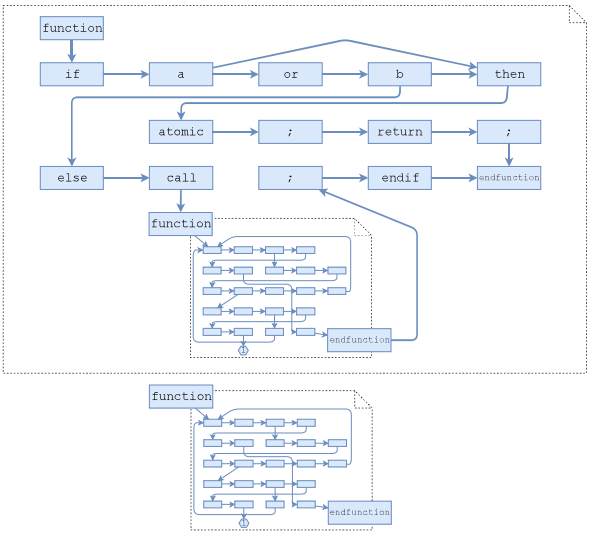


Figure 3: The GCFG of the program in Figure 2.

meaning (semantics) of any piece of smart contract code. This is achieved by means of a virtual machine.

Ethereum Virtual Machine (EVM). The EVM is the runtime environment for Ethereum programs (smart contracts). It is a Turing-complete stack machine programmable with a formally defined Bytecode format [49]. Every node in the Ethereum network runs an instance of the EVM. This ensures that all nodes are in consensus about the results of any transaction (function call). However, the downside is that every function call has to be executed by every node. Therefore, there are considerable costs associated with computations in smart contracts and the network might be attacked by spammers who intend to drain its computational power. Ethereum addresses this problem using the concept of Gas.

Gas. Every operation in the EVM has an associated cost, in Ether, which is roughly correlated with the amount of computational power it uses [49]. This cost is called gas. When a user calls a function of a smart contract, she has to pay the total gas cost associated with the operations executed by the contract. Similarly, when a contract calls a function in another contract, it should use the gas it has received from the original caller to pay for the operations executed by the other contract. The user includes a prepayment (deposit) of gas with her transaction. If the paid deposit is insufficient, the transaction will be rejected and the affected contracts will be reverted to their original state [9, 49]. If the transaction uses less gas than the deposit, the remaining gas is reimbursed.

Attacks on Ethereum Contracts. There are a variety of security vulnerabilities and possible attacks on Ethereum contracts [2]. A well-known family of these attacks are called gas limit attacks. Every Ethereum block is limited to handling at most a specific amount of computation, known as the gas limit. If a function call consumes more gas than the gas limit, it will fail and the consequences are often mishandled by smart contract programs [21]. Hence, a common best-practice is to avoid writing codes that can have an unbounded runtime. In practice, given the cost of gas, one should aim to develop smart contracts that execute as little computation as possible. Many real-world smart contracts do not even have loops [21].

In this paper, we consider two smart contract programming languages. We provide a short introduction to each of them.

Solidity. Solidity is a programming language for writing Ethereum smart contracts [50]. It was developed by the Ethereum team in 2014 and is currently the most widely-used smart contract programming language. Solidity aims to provide the programmer with all the usual functionality of a general-purpose language like C++.

Vyper. Vyper is the newest language of the Ethereum foundation [27]. It is a python-like scripting language whose goal is to provide a simple way of writing secure real-world smart contracts, by disallowing vulnerable functionality, and hence losing Turing-completeness, in exchange for more security [27]. One of the decisions by the designers of Vyper was to disallow the rarely-used functionality of infinite loops and recursion, in order to avoid gas limit attacks. Hence, all Vyper smart contracts are simple programs [27].

Types of Termination. A function in a smart contract can terminate in a number of ways. On Ethereum, the possibilities are [9, 49]:

(i) *Return*: As in classical programs, the function can terminate by returning control to its parent function.
(ii) *Revert*: The function can terminate by canceling the entirety of the current transaction, e.g. when an error occurs and the whole transaction must be rolled back. In this case, all the changes made by the current transaction, including those made by other functions, possibly even by other contracts, will be reverted. There are essentially two types of reversion: the programmer can choose to either refund the remaining gas after reversion or to burn it. These correspond to the `require` and `assert` keywords in Solidity.
(iii) *Self-destruct*: Finally, a function can terminate by destructing the current contract, making it unusable in the future, i.e. the functions of the contract will no longer be callable by anyone. Self-destruction is usually used when a contract reaches its expiration and is no longer useful, or when serious errors or security problems happen and it is necessary to stop any further interaction with the contract. In this case, the balance of the contract will be transferred to a predefined recipient.

Modeling Vyper and Solidity Contracts in ES. In our language, ES, we use `return` to model termination by returning and $\texttt{exit}_1$, $\texttt{exit}_2$ and $\texttt{exit}_3$ to model the other types of termination. A Solidity or Vyper `while(`ϕ`)` loop can simply be modeled by an ES `loop` whose body begins with `if(!`ϕ`) then break; endif`. Other types of loops, such as `for` can be modeled similarly. Hence, to prove that Solidity and Vyper smart contracts have CFGs (or GCFGs) of bounded treewidth, it suffices to show the same fact for ES.

Remark. In this work, we are considering structured, i.e. `goto`-free, programs. Therefore, we do not consider Solidity programs that include the so-called Solidity assembly code. It is well-known that one can write assembly codes that have arbitrarily large treewidth [32].

2.3 Motivating Examples

Tree decompositions and treewidth are formally defined in the next section. In this section, we provide some motivating examples to illustrate the importance of treewidth boundedness.

Formal Analysis. Exploiting graph structures for obtaining faster analysis algorithms is a well-studied field [3]. Bounded treewidth is

one of the most widely-used structures and leads to efficient algorithms for many formal analysis problems [29]. The usual approach to formal program analysis is to write the desired property of the program in a specification language or logic. For example, in case of smart contracts, a desired property might be that one party cannot cause a self-destruction of the contract if another party opposes it. As another example, to avoid the DAO attack, we can specify the property of avoiding the reentrancy vulnerability over all runs of the contract. Two of the most commonly-used formal languages for specifying desired properties are the μ-calculus and the Monadic Second Order Logic (MSO). There is no known polynomial-time algorithm for the problem of μ-calculus model checking, i.e. checking whether a given program/contract satisfies a property specified in μ-calculus, but the problem can be solved in linear time if the CFG has constant treewidth [41]. Similarly, model checking MSO properties is NP-hard in general, but can be done in linear time if the underlying graph has constant treewidth [36].

Compiler Optimization. A classical and well-studied problem in compiler optimization is that of register allocation [34], i.e. assigning program variables to a limited number of registers in an optimal manner. Register allocation is one of the most important stages for optimizing several typical goals, such as energy efficiency, code size and execution speed [34]. This problem is usually reduced to graph coloring, which is NP-hard even for 3 colors (equivalent to 3 registers) [47]. However, if the CFG has constant treewidth, then register allocation can be solved in polynomial time [33, 47]. In case of smart contracts, such optimizations at compile time can significantly reduce the gas costs and the overall energy that is used by the network to run a contract.

Quantitative Analysis. In contrast with classical verification, which classifies a program as either correct or incorrect, quantitative analysis assigns a value to every run of the program that quantifies the cost/revenue generated by that run [17]. In case of smart contracts, this value can naturally model financial gains or losses of a party in the contract, or the amount of gas/energy used by the contract. Hence, quantitative analysis of smart contracts is a natural and important problem [15]. In [17], it was shown that treewidth can help significantly in speeding up the computation of several major notions of quantitative analysis. Therefore, treewidth boundedness leads to much faster algorithms for analyzing the economic effects of a smart contract.

2.4 Tree Decompositions and Treewidth

In this section, we provide a succinct review of the notions of treewidth and tree decomposition. For a more in-depth treatment see [23]. Treewidth is one of the most widely used parameterizations for graph problems. Intuitively, the treewidth of a graph is a measure of how "tree-like" the graph is. However, the formal definition of treewidth is based on tree decompositions.

Tree Decompositions. Consider a graph $G = (V, E)$. A tree decomposition of G is a pair $(T, \{X_t | t \in T\})$ where T is a tree and every node t of T is labeled by a subset $X_t \subseteq V$ of vertices of G, such that the following conditions are satisfied:

- Every vertex $v \in V$ must appear in at least one X_t, i.e. $\cup_{t \in T} X_t = V$;
- For every edge $\{u, v\} \in E$, there must exist an X_t containing both u and v, i.e. $\forall e \in E\ \exists t \in T\ e \subseteq X_t$;
- For each vertex $v \in V$, the set $T^v = \{t \in T | t \in X_t\}$ must be a connected subtree of T. Note that T^v is the set of all nodes of T that contain v in their corresponding X_t. Hence, this condition means that every vertex should appear in a connected subtree of T.

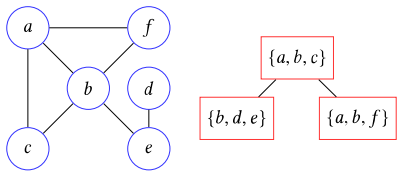


Figure 4: A graph G (left) and one of its optimal tree decompositions $(T, \{X_t\})$ (right).

To avoid confusion, we reserve the word "vertex" for vertices of G and use the word "node" to refer to vertices of the tree T. Also, we call each X_t a "bag".

Treewidth. The width of a tree decomposition $(T, \{X_t\})$ is defined as the size of the largest bag minus 1, i.e. $w(T, \{X_t\}) := \max_{t \in T} |X_t| - 1$. The treewidth $tw(G)$ of a graph G is defined as the smallest width among all tree decompositions of G.

Example. Figure 4 shows a graph G and one of its tree decompositions. This decomposition has a width of 2. It is easy to verify that G cannot have a tree decomposition of width 1. Hence, this tree decomposition is optimal and the treewidth of G is 2.

Dynamic Programming Algorithms. The significance of treewidth and tree decompositions in algorithm design stems from the fact that many hard graph problems can be solved in polynomial (often linear) time by performing a bottom-up dynamic programming on the tree decomposition, in essentially the same manner that is employed for solving problem on trees [6, 7, 23, 24]. A main concept in these algorithms is that one can associate a subgraph of G to every node of T. To do so, we fix an arbitrary node r as the root of T. Then, for each node $t \in T$, we let its corresponding subgraph G_t consist of all the vertices that appear in the bags of the subtree of T rooted at t, i.e. either in X_t or in the bags of its descendant nodes. Similarly, the edges of G_t are those edges of G that have both their endpoints appearing together in some bag in the subtree rooted at t. Hence, $G_r = G$ and if a node t has children $t_1, t_2, \ldots, t_k$, then for all t_i, we have $G_{t_i} \subseteq G_t$. The basic idea is then to compute the answer(s) to the problem at G_t by means of divide-and-conquer based on the answer(s) at G_{t_i}'s.

We now provide a different but equivalent formulation of the notion of treewidth as in [47]. We will make use of both formulations in our proofs in Sections 3 and 4.

Listings. Given a graph $G = (V, E)$, a listing L is simply a permutation of the vertices of G, i.e. a sequence of elements of V in which every $v \in V$ appears exactly once.

Separators. Given a graph $G = (V, E)$, a listing L and a vertex $v \in V$, let $l(v)$ be the set of all vertices that appear before v in the listing L and $r(v)$ be the set of all vertices that appear after v. Then the separator of v is the set of all vertices in $l(v)$ that can be reached from v using a path whose internal vertices are all in $r(v)$. We use

the notation S_v^L, or simply S_v when L is clear from the context, to denote the separator of v.

Complexity of Listings and Graphs. The complexity of a listing L is defined as the size of its largest separator, i.e. $c(L) := \max_{v \in V} |S_v^L|$. The complexity $c(G)$ of a graph G is defined as the minimum complexity among all its listings.

Example. Consider the graph in Figure 4 together with the listing $L = \langle a, b, c, d, e, f \rangle$. We have the following separators: $S_a = \emptyset, S_b = \{a\}, S_c = \{a, b\}, S_d = \{b\}, S_e = \{b, d\}, S_f = \{a, b\}$. Hence, L has complexity 2. One can also verify that G has no listing of a lower complexity, hence $c(G) = 2$.

We now review some previous lemmas and results.

Lemma 2.1. *For every graph G, we have $c(G) = tw(G)$, i.e. the complexity of a graph is the same as its treewidth [25]. Moreover, there is an algorithm to obtain a tree decomposition of width k from a listing of complexity k in linear time [47].*

Lemma 2.2. *[Contraction Lemma] Consider a graph $G = (V, E)$ of treewidth k and an edge $\{u, v\} \in E$. Let G' be the graph obtained by contracting $\{u, v\}$ in G. Then $tw(G') \leq tw(G)$. Moreover, there is a linear-time algorithm that given a listing of G with complexity k, produces a listing of G' with complexity at most k [47].*

Treewidth of Control Flow Graphs. In [47], it was shown that CFGs of goto-free Algol and Pascal programs have a treewidth of at most 3, while C programs have a treewidth of at most 6. In [32] it was shown that one can write Java programs with arbitrarily large treewidth, but real-world Java programs typically have a treewidth of 2 or 3. In [8] a similar result was obtained for Ada programs.

Remark. Note that CFGs are directed graphs, but the directions of edges are unimportant when computing tree decompositions and treewidth, and are therefore ignored in the rest of this paper.

3 INTRAPROCEDURAL TREEWIDTH OF SMART CONTRACTS

In this section, we consider the CFGs of ES programs and show that they always have bounded treewidth. As argued in Section 2.2, Solidity and Vyper programs can be modeled in ES and hence a treewidth boundedness result for ES naturally extends to contracts written in these languages. We start by enumerating the possible neighbors of every vertex in the CFG of an ES function. Then, closely following the construction in [47], we provide a natural way of obtaining a listing from an ES program. Finally, we compute an upperbound for the complexity of this listing, hence bounding the treewidth using Lemma 2.1.

Neighbors of a Vertex. Consider the CFG G_f of an ES function f and let v be a vertex in G_f. We use v^- (resp. v^+) to denote the predecessor (resp. successor) of v in G_f. If there are more than one successor or predecessor, we will take the one that is not inside the block of v. So, a `return`$^+$ is the semicolon following a `return` and a `loop`$^-$ is the vertex before that `loop` (and not the last vertex inside the block of the `loop`). Moreover, if v corresponds to an atomic boolean variable, then we use T_v (resp. F_v) to denote the vertex of the CFG that will be visited after v if its value is True (resp. False). Table 1 lists potential neighbors of a vertex based on its type.

Type of the Vertex v	Potential Neighbors in CFG
`function`	v^+
`endfunction`	v^-, `return`$^+$ vertices
`if`	v^+, v^-
`then`	v^+, atomic boolean variables in the corresponding `if` condition
`else`	v^+, atomic boolean variables in the corresponding `if` condition
`endif`	v^+, last vertex in the corresponding `then` block, last vertex in the corresponding `else` block
`loop`	v^+, v^-, `continue`$^+$ vertices referring to v, last vertex in the block of v
`endloop`	v^+, `break`$^+$ vertices referring to the corresponding `loop`
`break`	v^+, v^-
`break`$^+$	v^-, the corresponding `endloop`
`continue`	v^+, v^-
`continue`$^+$	v^-, the corresponding `loop`
`return`	v^+, v^-
`return`$^+$	v^-, `endfunction`
`exit`$_i$	v^+, v^-
`exit`$_i^+$	v^-, hexagon node i
`call`	v^+, v^-
`call`$^+$	v^+, v^-
`atomic`	v^+, v^-
`atomic`$^+$	v^+, v^-
`and`	v^+, atomic boolean variables in v's left-hand-side expression
`or`	v^+, atomic boolean variables in v's left-hand-side expression
atomic boolean variable	v^-, T_v, F_v
hexagon node i	`exit`$_i^+$ vertices

Table 1: Potential neighbors of a vertex v in the CFG

Canonical Listing of an ES Function. Given an ES function f and its CFG G_f, the canonical listing $CL(f)$ is a listing that visits the vertices of G_f in the following recursive manner:

- If the function is of the form `function` f A `endfunction`, we first visit `endfunction`, followed by `function`, the hexagonal nodes $1, \ldots k$, and finally a recursive visiting of A.
- If A is of the form B ; C, we first visit the semicolon, followed by a recursive visit of B and then a recursive visit of C.
- If A is of the form B C, where B is a statement that has an "end" vertex, i.e. if B is either an `if-then(-else)-endif` statement or a `loop-endloop` statement, we first visit B and then C, both recursively[3].
- If A is of the form `if` B `then` C `else` D `endif`, we visit it in this order: `endif`, `if`, `then`, `else`, B, C, D. Where the visits to B, C and D are recursive. Similarly, if A is of the form `if` B `then` C `endif`, the visiting order would be `endif`, `if`, `then`, B, C.
- If A is of the form `loop` B `endloop`, we first visit `endloop`, followed by `loop` and a recursive visit of B.
- If A is a boolean expression of the form B `or` C, we first visit `or` and then visit B and C recursively. We do the same for B `and` C.
- If A is an atomic boolean variable or statement, we just visit it, i.e. add it to the listing.

[3]There might be several ways of writing A as B ; C or B C. In such cases, we take the one that leads to the shortest B.

Intuitively, we visit the parts of the program in a top-down fashion.

Separators. We now find the separators of every vertex in the CFG G_f with respect to the canonial listing $CL(f)$. The vertex endfunction appears in the beginning of the listing, so $S_{\texttt{endfunction}} = \emptyset$. Given that endfunction is the only vertex appearing before function, we have $S_{\texttt{function}} \subseteq \{\texttt{endfunction}\}$. For a word (vertex) v, we use $\underline{v}$ (resp. $\overline{v}$) to denote the word preceding (resp. succeeding) v in the ES program[4]. Similarly, if X is a set of words that appear consecutively in the program, we use $\underline{X}$ (resp. $\overline{X}$) to denote the word exactly before (resp. after) X. Now consider a vertex v,

- If v is an if, then the separator of v can include $\underline{v}$ and endif because they are connected to v in the CFG and appear before v in the listing. However, if v is inside a loop, the separator can also contain loop and endloop due to the possibility of existence of continue and break statements in the blocks of v. Similarly, exit and return statements make it possible for the separator to contain the hexagonal nodes $1, 2, \ldots, k$ and the vertex endfunction. Hence we have $|S_v| \leq k + 5$. Recall that k is the number of different exit types.
- The words then, else and endif have almost the same situation. If v is a vertex corresponding to the word then, we have $S_v \subseteq \{\texttt{if}, \texttt{endif}, \texttt{loop}, \texttt{endloop}, \texttt{endfunction}, 1, \ldots, k\}$. Also, if v is a vertex corresponding to the word else, then $S_v \subseteq \{\texttt{if}, \texttt{then}, \texttt{endif}, \texttt{loop}, \texttt{endloop}, \texttt{endfunction}, 1, \ldots, k\}$. For the case of endif, we have $S_v \subseteq \{\underline{\texttt{if}}, \texttt{loop}, \texttt{endloop}, \texttt{endfunction}, 1, \ldots, k\}$.
- If v is a loop, then $\underline{v} = v^-$, hence $\underline{v}$ is a neighbor of v in the CFG and appears before v in the listing. Therefore $\underline{v}$ is in the separator. So $S_v \subseteq \{\underline{v}, \texttt{endloop}, \texttt{endfunction}, 1, \ldots, k\}$.
- If v is an endloop and u is the corresponding loop, then $\underline{u} \in S_u$ and hence $\underline{u} \in S_v$. So we have $S_v \subseteq \{\underline{u}, \texttt{loop'}, \texttt{endloop'}, \texttt{endfunction}, 1, \ldots, k\}$. Here, loop'-endloop' is the higher level loop containing u and v (if such a loop exists).
- If v corresponds to either of atomic, break, continue, call or return, it is easy to see that $v^+ = \overline{v}$ and $v^- = \underline{v}$ and they are both visited before v. Hence, $S_v = \{v^+, v^-\}$.
- If v is a break$^+$, i.e. the semicolon after a break vertex v^-, then v^{--} appears before v in the listing and the path $v \to v^- \to v^{--}$ exists in the CFG. So if v^{--} exists, then $v^{--} \in S_v$. Hence, $S_v \subseteq \{v^{--}, \texttt{endloop}\}$, because in the CFG we jump straight to endloop after v. Similarly, we have $S_v \subseteq \{v^{--}, \texttt{loop}\}$ (resp. $S_v \subseteq \{v^{--}, \texttt{endfunction}\}$) if v is a continue$^+$ (resp return$^+$). However, if v is atomic$^+$ or call$^+$, then the execution of the function continues as usual after v, so $S_v \subseteq \{v^{--}, \texttt{loop}, \texttt{endloop}, \texttt{endfunction}, 1, \ldots, k\}$.
- If v is an and vertex in A and B, then it is easy to check that $S_v \subseteq \{\texttt{if}, \texttt{then}, \texttt{else}, \texttt{endif}, \underline{A}, \overline{B}\}$. The same holds if v is an or vertex. Similarly, if v is an atomic boolean variable, then $S_v \subseteq \{T_v, F_v, \underline{v}, \overline{v}\}$.
- Finally, if i is a hexagonal exit node, then $S_i \subseteq \{\texttt{function}, \texttt{endfunction}, 1, 2, \ldots, k\}$.

Hence, we have the following lemma:

Lemma 3.1. *For every ES function f with k exit types, the canonical listing $CL(f)$ of the CFG G_f has a complexity of at most $k + 6$.*

Proof. The cases above show that every vertex has a separator of size at most $k + 6$. The complexity of a listing is the size of the largest separator. Hence the desired result is obtained. □

Corollary 3.1. *The CFG of any ES function f with k exit types has a treewidth of at most $k + 6$.*

Proof. By applying Lemmas 3.1 and 2.1. □

Remark 1 (Sharpness). *The bound obtained in Corollary 3.1 is sharp, i.e. for every k, one can obtain an ES programs with k exit types and a treewidth of exactly $k + 6$, by ensuring that for any vertex type, there is a vertex v in the CFG whose separator S_v includes all the possible cases enumerated above.*

We are now ready for the main theorem of this section.

Theorem 3.1. *The CFG of every Solidity or Vyper smart contract has a treewidth of at most 9.*

Proof. As shown in Section 2.2, Solidity and Vyper smart contracts can be modeled by ES programs with $k = 3$ exit types, i.e. revert with gas refund, revert without gas refund and self-destruct. Note that returning control to the parent function is already modeled in ES and does not need a new exit type. Hence, the desired result is obtained by applying Corollary 3.1. Also note that, unlike ES, the conditional and loop structures in Solidity and Vyper need not have endif or endloop specifiers, e.g. a Solidity if block that contains a single statement does not need to be enclosed in braces. However, this is not an issue, given that one can simply contract the endloop and endif nodes without increasing the treewidth (Lemma 2.2). The same point is also applicable to then. □

Remark. Note that our approach is constructive and we provided a linear-time algorithm for obtaining a 9-complex listing by one pass over the code of the smart contract. This, together with the algorithm of Lemma 2.1, ensure that one can obtain a tree decomposition of width 9 from the contract code in linear time.

4 INTERPROCEDURAL TREEWIDTH OF SMART CONTRACTS

In this section, we consider the treewidth of GCFGs of smart contracts. If we are given a constant-width tree decomposition of the GCFG of a contract, then we can naturally apply dynamic programming algorithms for solving interprocedural (global) problems. Intuitively, given that CFGs of smart contracts have constant treewidth and GCFGs are obtained by piecing copies of CFGs together in a structured manner, it should come as no surprise that GCFGs have constant treewidth, too. We formally prove this in Theorem 4.1.

Given this treewidth boundedness result, the only remaining challenge for applying dynamic programming algorithms is the size of the GCFG and the resulting tree decomposition. For recursive programs, the GCFG is infinite. Fortunately, as mentioned in

[4]Note that these are not necessarily the same as v^- and v^+ which are defined using the CFG, not the order of the words in the ES program

Section 2.2, all Vyper smart contracts are simple programs. On the other hand, while recursion is possible in Solidity, our experimental results (Section 5) show that real-world Solidity contracts are simple programs as well. Unfortunately, even for simple programs, the GCFG, and hence its tree decompositions, can have exponential size with respect to the length of the program. We overcome this difficulty by designing a succinct representation of tree decompositions that (i) can be directly computed from the program code in linear time, and (ii) is compatible with bottom-up dynamic programming algorithms, allowing them to run in polynomial time (with respect to the length of the program, rather than the size of the GCFG) by eliminating unnecessary repeated computations. We now prove interprocedural treewidth boundedness.

Theorem 4.1. *The GCFG of every Solidity or Vyper smart contract has a treewidth of at most 10.*

Proof. Let C be a Solidity or Vyper smart contract with functions $f_1, f_2, \ldots, f_n$. Also, let G be the GCFG of C and G_i be the CFG of the function f_i. Theorem 3.1 guarantees that every G_i has a treewidth of at most 9. Let $\tau_i = (T_i, \{X_t | t \in T_i\})$ be a tree decomposition of G_i with width at most 9 and τ_i^+ be a tree decomposition of G_i, obtained by adding the `endfunction` vertex to every bag, i.e. $\tau_i^+ = (T_i, \{Y_t | t \in T\})$ where $Y_t = X_t \cup \{\texttt{endfunction}\}$. We show how to create a tree decomposition $\tau = (T, \{X_t | t \in T\})$ of width at most 10 of G using the τ_i's and τ_i^+'s. The process mimics the procedure for creating G using the G_i's. We start with an unexpanded graph G^0 and a tree decomposition τ^0 of G^0. In each step, we expand the `call` vertices in G^k to obtain G^{k+1}. We also create a new tree decomposition τ^{k+1}. The invariant satisfied by the algorithm is that for every k, the obtained τ^k is always a tree decomposition of G^k of width at most 10.

(1) Let $G^0 = \bigcup_{i=1}^n G_i$ and $\tau^0 = \bigcup_{i=1}^n \tau_i$. Also, set an arbitrary node in every connected component of τ^0 as the root.
(2) While there is an unexpanded `call` vertex in G^k:
   - Let $G^{k+1} = G^k$.
   - For every unexpanded call vertex of G^k of the form `call` f_i`;` appearing in the function f_j:
     - Expand the `call` in G^{k+1}.
     - Let t be a node in τ^k such that the bag X_t contains both the `call` and the semicolon following it. Note that such a node must exist because the `call` and the semicolon are neighbors in G^k and τ^k is a tree decomposition of G^k. Also, Let s be a node in τ_i^+ whose bag X_s contains the vertex `function`.
     - Create a new node t' in τ^{k+1}, connect it to t as a child, and let $X_{t'} = \{\texttt{call}, \texttt{;}, \texttt{function}, \texttt{endfunction}\}$. This bag contains two elements from X_t and two from X_s. We call t' an intermediary node.
     - Add a copy of τ_i^+ (rooted at s) to τ^{k+1}, i.e. let $\tau^{k+1} = \tau^{k+1} \sqcup \tau_i^+$, and connect t' to the node s in this copy such that t' is the parent and s is the child.

It is easy to check that the procedure above satisfies the invariant. Clearly, τ^0 is a tree decomposition of G^0 with width at most 9. In the process of expanding a function call in G^{k+1}, all the new vertices and edges between them are covered in τ^{k+1} by the new copy of τ_i^+, which has a width of at most 10. The edges from `call` to `function` and from `endfunction` to the semicolon appear in $X_{t'}$.

Finally, in case of simple programs, the process ends at some point. Hence there is a G^k such that $G^k = G$. So τ^k is the desired τ. In case of recursive programs, we have $G = \bigcup_{i=0}^{\infty} G^i$ and can hence define $\tau := \bigcup_{i=0}^{\infty} \tau^i$. This completes the proof. □

Succinct Representation of Interprocedural Tree Decompositions. We now consider simple programs only. Note that the tree decomposition τ created in the process above has a lot of redundancy. Basically, τ is obtained by piecing together the τ_i's, which are tree decompositions of the CFGs of the functions f_i, with many copies of the τ_i^+'s and intermediary nodes (the t' nodes in the proof above). We can eliminate this redundant copying without affecting the results of dynamic programming algorithms. Formally, let t be a node in τ_i^+ (or an intermediary node), then it is easy to verify that, by construction, every copy of t in τ has the same associated subgraph $G_t \subseteq G$ (up to the natural isomorphism) and the same bag X_t. Therefore, in a dynamic programming scheme, we can only compute the answers in one of the copies of G_t, and reuse them for all other copies. Equivalently, we can represent the rooted tree decomposition $\tau = (T, \{X_t | t \in T\})$ in a succinct manner $\tau^* = (T^*, \{X_{t^*} | t^* \in T^*\})$ by merging all copies of the same τ_i^+ (or intermediary node) into one. This of course means that the same node can now have several parents and T^* is a DAG, i.e. directed acyclic graph, instead of a rooted tree. However, the dynamic programming algorithms can be applied to τ^* in the same manner as in τ, i.e. in bottom-up order[5]. This automatically avoids the redundant and repetitive computations at every copy of τ_i^+.

Theorem 4.2. *The GCFG of every simple Vyper or Solidity smart contract has a tree decomposition of width at most 10 that is compatible with bottom-up dynamic programming algorithms and can be succinctly represented in linear size, with respect to the length of the contract. Moreover, this succinct representation can be obtained from the contract code in linear time.*

Proof. We take the tree decomposition τ and its succinct representation τ^* as described above. We have already shown the width and compatibility with bottom-up dynamic programming. We just need to prove that τ^* has linear size and can be obtained in linear time. Each of the one-function tree decompositions τ_i and τ_i^+ appear exactly once in τ^* and there are at most as many intermediary nodes as the number of `call` operations in the code. Hence, τ^* has linear size. Also, the same process that was used for obtaining τ in the proof of Theorem 4.1 can be applied to obtain τ^*, except that every `call` site should be expanded only once. Hence, τ^* can be computed in linear time. □

5 EXPERIMENTAL RESULTS

Our Tool. We implemented a tool in Python/C++ that gets a Solidity smart contract as input and outputs its canonical listing, tree decompositions of its intraprocedural CFGs (with width at most 9) and the succinctly-represented tree decomposition τ^* of its interprocedural GCFG (with width at most 10). Our tool works in linear

[5]Note that the simplicity assumption is indeed necessary. If the program is recursive, then T^* contains a cycle and is no longer a DAG. Hence, there is no bottom-up ordering of the nodes and dynamic programming algorithms cannot be applied.

time and is very efficient in practice. We use the ConsenSys Solidity parser [22] to obtain the CFGs.

Benchmarks. We used the contracts listed in the Etherscan database of verified Solidity source codes [46] as our benchmarks. Ethereum contracts are saved on the Blockchain in Bytecode format, but many users like to see the actual Solidity code of the contract. Hence, Etherscan allows programmers to publish the original Solidity code, then compiles it and verifies that the resulting bytecode is the same as the one published on the Blockchain. Hence, all of our benchmarks are real-world smart contracts that are currently deployed on the Ethereum Blockchain.

Number of Benchmarks. At the time of writing, there are just below 40,000 smart contracts listed in the Etherscan database. Of these, we ignored contracts that include assembly code and hence can have arbitrarily large treewidth, and the ones that produced compilation errors[6]. This left us with 36,764 benchmarks.

Runtime and Machine. We used an Intel Core i5-7200U 2.5GHz Processor running Ubuntu 18.04. We ran our approach on all 36,764 benchmarks. In all cases, our runtime was less than 0.1 seconds.

Intraprocedural Results. We found that for CFGs of real-world smart contracts, the treewidth bound 9 is never met. The highest width among the obtained tree decompositions was 6 and the average was 3.35. Figure 5 shows the number of contracts with each width and Figure 6 shows the distribution of contracts based on their size and width. Note that the vast majority of obtained tree decompositions have a width of 3 or 4.

Interprocedural Results. We found that all our benchmarks are simple programs and recursion is not used in real-world Solidity smart contracts. Similar to the previous case, the bound 10 was never met in practice. The highest width of a GCFG tree decomposition in our benchmarks was 7 and the average width was 3.65. Figure 7 shows the number of contracts with each interprocedural width and Figure 8 shows their distribution based on contract size and width. The vast majority of contracts have an interprocedural tree decomposition of width 3 or 4. However, note that in our construction of τ (Theorem 4.1), the intermediary nodes have a bag of size 4, so the tree decompositions have a width of at least 3.

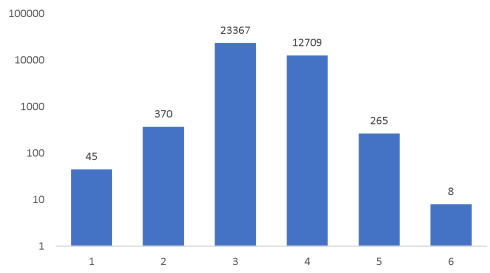


Figure 5: The number of benchmark contracts (y axis, log scale) for which the intraprocedural CFG tree decompositions obtained by our tool have a given width (x axis).

[6]This is possible because Solidity is not backwards-compatible and the codes could have been written in an earlier version of the language.

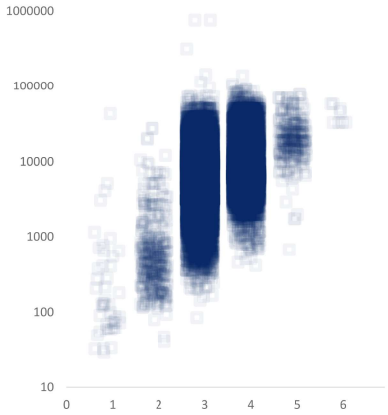


Figure 6: Distribution of benchmark contracts based on the width of the obtained tree decompositions of their intraprocedural CFGs (x axis) and size of the contract code in bytes (y axis, log scale).

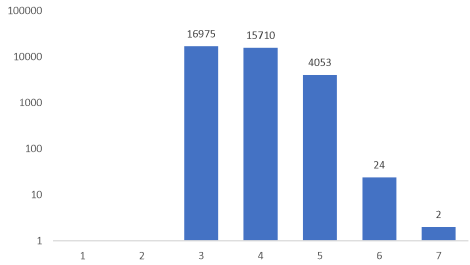


Figure 7: The number of benchmark contracts (y axis, log scale) for which the interprocedural GCFG tree decompositions obtained by our tool have a given width (x axis).

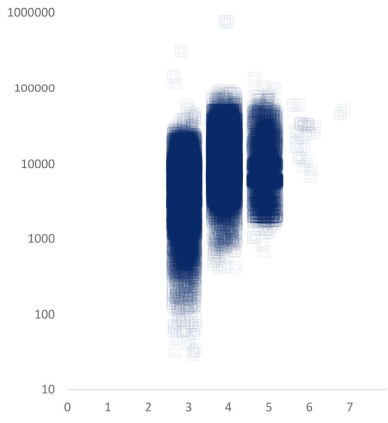


Figure 8: Distribution of benchmark contracts based on the width of the obtained tree decompositions of their interprocedural GCFGs (x axis) and size of the contract code in bytes (y axis, log scale).

6 CONCLUSION

In this paper, we showed that Ethereum smart contracts written in Solidity and Vyper have small treewidth. We obtained a sharp theoretical bound of 9 for the intraprocedural treewidth and 10 for the interprocedural case. We also reported on a tool we implemented for computing treewidth of Solidity contracts and provided experimental results that showed the treewidth of real-world contracts is often much smaller. We argued that the treewidth boundedness result can be exploited to obtain much faster algorithms for program analysis, model checking, compiler optimization and quantitative analysis of contracts. A natural next step would be to develop analysis and optimization tools for smart contracts using the currently-known faster algorithms that exploit treewidth, and also developing tools for obtaining treewidth of contracts written in other languages, especially Vyper.

ACKNOWLEDGMENTS

The research was partially supported by Vienna Science and Technology Fund (WWTF) Project ICT15-003, Austrian Science Fund (FWF) NFN Grant No S11407-N23 (RiSE/SHiNE), ERC Starting Grant (279307: Graph Games), and the IBM PhD Fellowship program.

REFERENCES

[1] Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.
[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *POST 2017*.
[3] Domagoj Babic. 2008. *Exploiting structure for scalable software verification.* Ph.D. Dissertation. University of British Columbia, Vancouver, Canada.
[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Fournet, et al. 2016. Formal verification of smart contracts: Short paper. In *PLAS 2016*.
[5] Bitcoin Wiki. 2018. Script. https://en.bitcoin.it/wiki/Script.
[6] Hans L Bodlaender. 1988. Dynamic programming on graphs with bounded treewidth. In *ICALP 1988*. Springer, 105–118.
[7] Hans L Bodlaender. 1994. A tourist guide through treewidth. *Acta cybernetica* 11, 1-2 (1994), 1.
[8] Bernd Burgstaller, Johann Blieberger, and Bernhard Scholz. 2004. On the tree width of ada programs. In *International Conference on Reliable Software Technologies*. Springer, 78–90.
[9] Vitalik Buterin. 2018. A Next Generation Smart Contract and Decentralized Application Platform. *Ethereum White Paper* (2018).
[10] Brad Calder and Dirk Grunwald. 1994. Reducing indirect function call overhead in C++ programs. In *POPL 1994*. ACM, 397–408.
[11] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL 2016*. 733–747.
[12] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Yaron Velner. 2018. Ergodic Mean-Payoff Games for the Analysis of Attacks in Crypto-Currencies. In *CONCUR 2018*. 11:1–11:17.
[13] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, and Andreas Pavlogiannis. 2019. Efficient Parameterized Algorithms for Data Packing. In *POPL 2019*.
[14] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2017. JTDec: A Tool for Tree Decompositions in Soot. In *ATVA 2017*. 59–66.
[15] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. 2018. Quantitative Analysis of Smart Contracts. In *ESOP 2018*. 739–767.
[16] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2018. Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components. *ACM Trans. Program. Lang. Syst.* 40, 3 (2018), 9:1–9:43.
[17] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2015. Faster algorithms for quantitative verification in constant treewidth graphs. In *CAV 2015*. 140–157.
[18] Usman Chohan. 2017. The Decentralized Autonomous Organization and Governance Issues. *Regulation of Financial Institutions eJournal* (2017).
[19] Coin Market Cap. 2018. Cryptocurrency Market Capitalizations. https://coinmarketcap.com/.
[20] ConsenSys. 2018. *Ethereum Smart Contract Best Practices.* https://consensys.github.io/smart-contract-best-practices/software_engineering/
[21] ConsenSys Diligence. 2018. Ethereum Smart Contract Best Practices - Known Attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/.
[22] ConsenSys Team. 2018. Solidity Parser. https://github.com/ConsenSys/solidity-parser.
[23] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. 2015. *Parameterized algorithms*. Springer.
[24] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Joham MM van Rooij, and Jakub Onufry Wojtaszczyk. 2011. Solving connectivity problems parameterized by treewidth in single exponential time. In *FOCS 2011*. 150–159.
[25] Nick D Dendris, Lefteris M Kirousis, and Dimitrios M Thilikos. 1997. Fugitive-search games on graphs and related parameters. *Theoretical Computer Science* 172, 1-2 (1997), 233–254.
[26] Edsger W Dijkstra. 1960. Recursive programming. *Numer. Math.* 2, 1 (1960), 312–318.
[27] Ethereum Foundation, Vitalik Buterin, et al. 2018. Vyper Language Documentation. https://vyper.readthedocs.io.
[28] Ethernodes. 2018. The Ethereum Node Explorer. https://www.ethernodes.org.
[29] Andrea Ferrara, Guoqiang Pan, and Moshe Y Vardi. 2005. Treewidth in verification: Local vs. global. In *LPAR 2005*. 489–503.
[30] Amir Kafshdar Goharshady, Ali Behrouz, and Krishnendu Chatterjee. 2018. Secure Credit Reporting on the Blockchain. In *IEEE International Conference on Blockchain*.
[31] Amir Kafshdar Goharshady and Fatemeh Mohammadi. 2017. A Short Note on Parameterized Computation of Network Reliability with respect to Treewidth. *arXiv preprint arXiv:1712.09692* (2017).
[32] Jens Gustedt, Ole A Mæhle, and Jan Arne Telle. 2002. The treewidth of Java programs. In *ALENEX 2002*. 86–97.
[33] Philipp Klaus Krause. 2013. Optimal register allocation in polynomial time. In *CC 2013*. 1–20.
[34] Philipp Klaus Krause. 2014. The complexity of register allocation. *Discrete Applied Mathematics* 168 (2014), 51–59.
[35] Neeraj Kumar. 2015. *Graph-theoretic Properties of Control Flow Graphs and Applications.* Master's thesis. University of Waterloo.
[36] Alexander Langer, Felix Reidl, Peter Rossmanith, and Somnath Sikdar. 2014. Practical algorithms for MSO model-checking on tree-decomposable graphs. *Computer Science Review* 13 (2014), 39–74.
[37] Janusz Laski and William Stanley. 2009. *Software verification and analysis: An integrated, hands-on approach.* Springer Science & Business Media.
[38] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *CCS 2016*. ACM, 254–269.
[39] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
[40] Zeinab Nehai, Pierre-Yves Piriou, and Frederic Daumas. 2018. Model-Checking of Smart Contracts. In *IEEE International Conference on Blockchain*. IEEE, 980–987.
[41] Jan Obdržálek. 2003. Fast mu-calculus model checking when tree-width is bounded. In *CAV 2003*. Springer, 80–92.
[42] Neil Robertson and Paul D Seymour. 1984. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B* 36, 1 (1984), 49–64.
[43] Barbara G Ryder. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 3 (1979), 216–226.
[44] Lakshmi Siva Sankar, M Sindhu, and M Sethumadhavan. 2017. Survey of consensus protocols on blockchain applications. In *ICACCS 2017*. IEEE, 1–5.
[45] Amitabha Sanyal, Bageshri Sathe, and Uday Khedker. 2009. *Data flow analysis: theory and practice.* CRC Press.
[46] Matthew Tan, Wee Chuan, Jann Yik, et al. 2018. Etherscan: The Ethereum Block Explorer. https://etherscan.io/.
[47] Mikkel Thorup. 1998. All structured programs have small tree width and good register allocation. *Information and Computation* 142, 2 (1998), 159–181.
[48] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS 2018*. 67–82.
[49] Gavin Wood. 2018. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Yellow Paper* (2018).
[50] Gavin Wood et al. 2018. Solidity Language Documentation. https://solidity.readthedocs.io/en/v0.4.24/.
[51] David Zimbeck, Sean Donato, et al. 2018. Bithalo, Mother of Smart Contracts and a Decentralized Market for Everything. http://bithalo.org.